Data Placement Strategies for Heterogeneous and Non-Volatile Memories in High Performance Computing

Andrès Rubio Proaño

To cite this version:

Andrès Rubio Proaño. Data Placement Strategies for Heterogeneous and Non-Volatile Memories in High Performance Computing. Distributed, Parallel, and Cluster Computing [cs.DC]. Université de Bordeaux, 2021. English. NNT : 2021BORD0224 . tel-03431281

HAL Id: tel-03431281
https://theses.hal.science/tel-03431281

Submitted on 16 Nov 2021

THÈSE PRÉSENTÉE

POUR OBTENIR LE GRADE DE

DOCTEUR

DE L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

INFORMATIQUE

Par **Andrès RUBIO PROAÑO**

Stratégies de Placement de Données pour Mémoires Hétérogènes et Non-Volatiles en Calcul Haute Performance

Sous la direction de : **Brice GOGLIN**

Soutenue le 7 Octobre 2021

Membres du jury:

Mme Rosa Badia ..	Directrice de recherche, Barcelona Supercomputing Center	Examinatrice
M. Thierry Gautier	Chargé de Recherche, Inria	Rapporteur
M. Brice Goglin ..	Directeur de recherche, Inria / LaBRI	Directeur
M. Raymond Namyst	Professeur, Université de Bordeaux	Président
M. François Trahay	Maître de conférence, Institut Polytechnique de Paris ..	Rapporteur

Titre Stratégies de placement de données pour les systèmes à mémoires hétérogènes dans le calcul haute performance.

Résumé Les systèmes mémoire des plates-formes de calcul haute performance ont subi des changements majeurs ces dernières années. En plus de la mémoire principale, du stockage et de plusieurs niveaux de caches, les serveurs sont à accès non-uniforme (NUMA) et peuvent disposer de plusieurs types de mémoire. Par exemple, les mémoires à haut débit (HBM) embarquées dans le processeur ainsi que les mémoires non volatiles (NVDIMM) ont été introduites dans la hiérarchie. Ces changements sont nécessaires pour rapprocher les données du traitement et donc avoir de meilleures performances. Cependant, ils obligent les développeurs à adapter leurs applications pour fonctionner correctement sur ces différents systèmes hétérogènes, ce qui rend le développement beaucoup plus complexe. En pratique, le simple fait de décider d'allouer un tampon de données sur le bon type de mémoire dans ces systèmes hétérogènes est difficile et critique pour les performances de l'application.

Cette thèse a été réalisée à Inria Bordeaux - Sud-Ouest et au LaBRI. Après avoir présenté l'état de l'art des architectures mémoire, nous avons caractérisés les différents types de mémoires à l'aide d'attributs simples. Nous avons fourni une interface que la bibliothèque hwloc expose aux applications pour comprendre l'organisation mémoire et allouer des tampons. Nous avons ensuite proposé une méthodologie pour que les développeurs puissent adapter leurs applications à l'utilisation appropriée de systèmes à mémoire hétérogènes. Comme l'accès à différents plates-formes hétérogènes n'est pas toujours possible, nous avons identifiés de nombreuses stratégies permettant la simulation des performances de mémoire hétérogène, et l'émulation de topologie de la mémoire différentes. Enfin, nous avons conçu une stratégie visant à faciliter le partage des plates-formes à mémoire non-volatile et hétérogènes entre des tâches HPC co-exécutées sur les mêmes serveurs.

Mots-clés Calcul haute performance, systèmes à mémoire hétérogène, modèles de programmation parallèle, programmation par tâches, calcul distribué, supports d'exécution

Laboratoire d'accueil Inria Bordeaux Sud-Ouest, 200 Avenue de la Vieille Tour, 33400 Talence

Title Data-Placement Strategies for Heterogenous Memory Systems in High Performance Computing.

Abstract Memory systems in High-Performance Computing (HPC) systems have undergone major changes in recent years. Beside main memory, storage and multiple levels of caches, servers come with non-uniform memory access (NUMA) and may contain different kinds of memory. For instance, high bandwidth memory (HBM) embedded on the processor package and non-volatile memory (NVDIMM) have been introduced into the hierarchy. These changes are necessary to bring the data closer and closer to processing and therefore have better performance. However, they require developers to adapt their applications to work properly on different heterogeneous memory systems, causing software development to become much more complex. In practice, the simple fact of deciding to allocate a data buffer on the appropriate memory in a heterogeneous system becomes difficult and critical to application performance.

This thesis has been carried out at Inria Bordeaux - Sud-Ouest and LaBRI. After a presentation of the state of the art of memory architectures, we have characterised the memories through simple attributes. We have provided an interface that the hwloc library exposes to applications to understand the memory topology and allocate buffers. Then, we proposed a strategy to help developers adapt their applications for the proper use of heterogeneous memory systems. As accessing different heterogeneous platforms is not always possible, we identify several ways to simulate the performance of heterogeneous memory and to emulate different memory topologies. Finally, we built a strategy that eases the sharing of platforms with heterogeneous and non-volatile memory between HPC tasks co-scheduled on the same nodes.

Keywords High performance computing, heterogeneous memory systems, parallel programming models, task-based programming, distributed computing, run-time systems

Hosting Laboratory Inria Bordeaux Sud-Ouest, 200 Avenue de la Vieille Tour, 33400 Talence

Título Estrategias de emplazamiento de datos para sistemas de memoria heterogéneos en computación de alto rendimiento.

Resumen Los sistemas de memoria en los sistemas de Computación de Alto Rendimiento han experimentado cambios importantes en los últimos años. Además de la memoria principal, el almacenamiento y varios niveles de cachés, los servidores vienen con acceso a memoria no uniforme (NUMA) y pueden contener diferentes tipos de memoria. Por ejemplo, la memoria de alto ancho de banda (HBM) incorporada en el paquete del procesador y la memoria no volátil (NVDIMM) se han introducido en la jerarquía. Estos cambios son necesarios para acercar cada vez más los datos al procesamiento y, por lo tanto, tener un mejor rendimiento. Sin embargo, requieren que los desarrolladores adapten sus aplicaciones para que funcionen correctamente en diferentes sistemas de memoria heterogéneos, lo que hace que el desarrollo de software se vuelva mucho más complejo. En la práctica, el simple hecho de decidir asignar un búfer de datos en la memoria adecuada en un sistema heterogéneo se vuelve difícil y crítico para el rendimiento de la aplicación.

Esta tesis se ha realizado en Inria Bordeaux - Sud-Ouest y LaBRI. Después de una presentación del estado del arte de las arquitecturas de memoria, hemos caracterizado las memorias a través de atributos simples. Hemos proporcionado una interfaz que la biblioteca hwloc expone a las aplicaciones para comprender la topología de la memoria y asignar búferes. Luego, propusimos una estrategia para ayudar a los desarrolladores a adaptar sus aplicaciones para el uso adecuado de sistemas de memoria heterogéneos. Dado que no siempre es posible acceder a diferentes plataformas heterogéneas, identificamos varias formas de simular el rendimiento de la memoria heterogénea y de emular diferentes topologías de memoria. Finalmente, creamos una estrategia que facilita el intercambio de plataformas con memoria heterogénea y no volátil entre tareas de HPC coprogramadas en los mismos nodos.

Palabras Clave Computación de alto rendimiento, sistemas heterogéneos de memoria, modelos de programación paralela, programación basada en tareas, computación distribuida, sistemas de tiempo de ejecución

Laboratorio Antitrión Inria Bordeaux Sud-Ouest, 200 Avenue de la Vieille Tour, 33400 Talence

Acknowledgments

I begin by thanking Jesus for his wonderful work on the cross, for allowing me to have an unforgettable experience during these three years of the thesis, and for putting the right people in order to finish this process successfully.

I thank the members of the jury who have taken the time to read my thesis. Thanks to Thierry and François for having read the manuscript in a very attentive way. To Rosa and Raymond for their valuable input as examiners.

I have no way of thanking Brice Goglin who has not only guided me in the completion of the thesis but has also been attentive at each stage of my adaptation in France. Thank you for all your help and for all that I have been able to learn from you.

I thank my beautiful and beloved wife Eilen Gordillo Proaño for accompanying me during these three years of the thesis. You could even be my co-worker during the lockdown. I love you and I will always love you.

I thank all my colleagues from the TADaaM team, Emmanuel, François, Alexandre, Francielli, Guillaume A and Guillaume. To my colleagues from the open space Valentin Honoré, Nicolas Vidal, Valentin Hoyet, Adrien Guilbaud, Florian Reynier, Philippe Swartvagher, Clément G, Clément B, Luan and Alexis. To my colleagues at HiePACS Alena and Esragul.

Thanks to the support of my parents, Mercedes Proaño and Xavier Rubio. I have no way of paying you for everything you have done for me. To my brother José Rubio Proaño and my sister-in-law María José Alcivar for all your love. To my in-laws Freddy Gordillo and Nery Proaño for always being aware of us. To my sister-in-law Valeska Gordillo for always brightening our days. To my siblings-in-law Ámbar Gordillo, Thiago Gordillo and Esteban Pérez. And to my beautiful niece, Agustina Pérez Gordillo.

To my beautiful Rubio Villegas family, especially my beloved grandparents Miguel Rubio Arteaga and Rosa Villegas Chávez who are in the presence of God and who have always driven me to be better.

To my beautiful Proaño Durán family, especially my beloved grandparents José Proaño Guevara and Luz Durán Merino for being that instrument to reach the feet of Christ.

Résumé étendu en français

L'humanité est confrontée au problème de créer une civilisation plus intelligente. Pour y parvenir, il est nécessaire de comprendre et de résoudre différents problèmes sociaux et scientifiques qui nous permettent d'avoir de nouveaux niveaux d'efficacité et d'optimisation. Ces problèmes nécessitent généralement de résoudre des tâches trop complexes. Par exemple, simuler la dynamique moléculaire pour créer de nouveaux matériaux.

En ce sens, le calcul haute performance est devenu un élément clé pour résoudre plusieurs de ces problèmes à travers des modèles, des simulations et des analyses. Mais pour devenir de plus en plus efficaces, ces systèmes doivent constamment évoluer.

Aujourd'hui, la tendance est de se rapprocher de l'exascale dans les systèmes et ainsi de résoudre les problèmes beaucoup plus gros et plus rapidement. Cependant, pour atteindre cet objectif, nous devons tout améliorer, du matériel aux applications. C'est là que l'hétérogénéité devient l'un des facteurs les plus récents permettant des systèmes plus rapides. Elle permet de travailler avec différents types d'applications où il peut être combiné, des processeurs pour les ordinateurs petits/complexes et des GPU pour les ordinateurs grands/simples. L'hétérogénéité n'a pas seulement approchée les processeurs, mais ces dernières années, elle a commencé à apparaître dans les systèmes de mémoire.

Actuellement, nous pouvons déjà trouver des systèmes de mémoire qui contiennent plusieurs types de mémoire coexistants. Les systèmes de mémoire dans le calcul haute performance (HPC) ont subi des changements majeurs ces dernières années. Outre la mémoire principale, le stockage et plusieurs niveaux de cache, les serveurs sont livrés avec un accès mémoire non uniforme (NUMA) et peuvent contenir différents types de mémoire. Par exemple, la mémoire à bande passante élevée (HBM) intégrée au processeur et la mémoire non volatile (NVDIMM) ont été introduites dans la hiérarchie.

Ces changements sont nécessaires pour rapprocher de plus en plus les données du traitement et donc avoir de meilleures performances. Cependant, ce saut nécessite que les développeurs prennent en compte le fait qu'ils doivent développer ou modifier des applications pour s'adapter au système de mémoire hétérogène. Cela pose la question principale qui est de savoir où allouer mes objets mémoire.

Avant de répondre à la question principale, il fallait d'abord être capable d'identifier quel type de mémoire nous avons dans le matériel. Comme mentionné, la gestion de la mémoire dans HPC devient de plus en plus difficile en raison de l'hétérogénéité réelle du système de mémoire. En ce sens, nous considérons que pour mieux prendre en charge ces nouveaux systèmes de mémoire, il est crucial d'identifier les types de mémoire et d'exposer leurs caractéristiques, afin que les couches logicielles supérieures puissent avoir une idée de l'endroit où allouer les tampons critiques pour les performances. Les systèmes de mémoire hétérogènes ne sont pas de forme unique et varient d'une plate-forme à l'autre, maintenant toujours une interaction d'au moins deux acteurs de mémoire différents, c'est-à-dire des systèmes qui combinent par example, HBM + DRAM.

En fait, identifier les types de mémoire consiste à comprendre quels nœuds NUMA sont de quel type. Pour ce faire, nous considérons que le système de mémoire est un élément clé dans la prise de conscience de la topologie pour avoir une allocation appropriée des tampons mémoire, et pour cela, chaque type de mémoire peut être caractérisé par certains attributs de mémoire tels que la bande passante, la latence, capacité, persistance, énergie, etc. Ces attributs nous permettent non seulement d'identifier le type de mémoire mais nous permettent également d'avoir un ordre sur le système de mémoire hétérogène. Dans ce travail, nous assignons un ensemble d'attributs à chaque dispositif mémoire, compte tenu de l'hétérogénéité du système. Ces attributs nous aident à les classer en fonction des métriques intuitives déjà mentionnées pour aider à la sélection du bon périphérique de mémoire en fonction d'un cas d'utilisation. Avec ces classements des périphériques mémoire disponibles, une application peut le choisir pour allouer ses données.

Ensuite, pour ouvrir ces idées aux développeurs, nous avons étendu une interface de programmation de la bibliothèque hwloc qui permet de récupérer la meilleure cible mémoire en fonction de l'attribut que l'on se demande d'utiliser. Il comprend des fonctions qui nous permettent d'identifier la meilleure cible mémoire étant donné un attribut mémoire et un initiateur (ensemble de cœurs accédant aux données). Il permet également de récupérer des informations sur les valeurs d'attributs, et il nous donne la possibilité d'ajouter nos propres attributs de mémoire.

Les valeurs d'attributs proviennent de deux sources principales. Le premier, correspond à l'utilisation des tables matérielles ACPI HMAT, dans lesquelles nous pouvons trouver des informations relatives à la localité, la latence et la bande passante. Cependant, cette table n'est pas encore implémentée dans tous les systèmes. L'alternative, la plus utilisée pour le moment, consiste à mesurer expérimentalement les valeurs des attributs. Cela nous permet de caractériser les performances matérielles mais surtout d'obtenir un ordre (étant donné un attribut) sur un système de mémoire hétérogène.

Une conséquence malheureuse de l'évolution et de l'hétérogénéité croissante des systèmes mémoire est la nécessité d'adapter les applications HPC afin qu'elles puissent exploiter correctement le système mémoire. Une première étape a été effectuée lors de l'identification et de la caractérisation du système de mémoire, cependant, cela ne répond pas aux considérations que le développeur devrait avoir lorsqu'il travaille avec une application avec des charges gourmandes en mémoire. En fait, comment le développeur peut-il savoir si une application ou ses buffers internes ont une certaine affinité avec un attribut mémoire ? En d'autres termes, pour savoir si l'attribut est critique pour la performance. Les applications HPC sont conçues pour prendre en charge un type de mémoire, cependant, nous considérons qu'étant donné que les applications peuvent être sensibles à la latence, la bande passante ou la capacité, il est important d'allouer au moins leurs principaux tampons au bon endroit. Nous proposons une stratégie qui met en œuvre des étapes pour fournir aux développeurs un environnement à haute productivité pour prendre en charge des systèmes de mémoire hétérogènes de manière portable. Dans cette stratégie, nous considérons qu'avant d'effectuer une demande d'allocation, nous avons besoin d'une étape intermédiaire pour déterminer la sensibilité de l'application et/ou de leurs buffers pour finalement obtenir un critère d'allocation. Nous considérons qu'il existe des développeurs très expérimentés qui sont capables de deviner si les performances d'accès à une zone sont plutôt limitée par la latence ou la bande passante, tout cela grâce à des années de travail d'optimisation de cache, de préchargement d'affinité, de tuilage, etc., un niveau de subjectivité basé sur l'expérience d'un individu. Compte tenu de cela, un cadre plus productif est nécessaire pour les non-experts en architectures matérielles et en optimisation de code.

Ensuite, nous considérons que nous devons analyser le comportement des applications pour déterminer la sensibilité des applications et de leurs buffers internes. Nous avons identifié trois options principales pour déterminer la sensibilité d'une application en évaluant, en profilant et en effectuant une analyse de code statique.

Avec le benchmarking, nous considérons que nous pouvons identifier la sensibilité d'une application en comparant différentes exécutions des applications en forçant l'ensemble du processus sur différents types de mémoire ou en forçcant chacun des tampons sur différents types de mémoire. Cette approche pour déterminer les sensibilités est encore largement utilisée dans la littérature pour montrer la sensibilité de certains repères à certains attributs. Cependant, la complexité des applications pourrait imposer de nombreuses exécutions différentes. Pour cette raison, nous étudions également le profilage.

Le profilage est une stratégie plus complexe qui effectue une analyse de l'exécution à l'aide de compteurs matériels et/ou d'instrumentation pour identifier en détail les problèmes liés à la mémoire tels que les goulots d'étranglement, les points chauds, etc. Il peut également être utilisé pour déterminer la sensibilité d'une application dans laquelle selon le profileur, nous pouvons avoir des métriques pertinentes qui nous donnent des indices que les applications pourraient avoir une certaine sensibilité en termes de latence ou de bande passante. De plus il est possible d'identifier très facilement les principaux tampons de l'application qui rencontrent des problèmes et d'identifier ces tampons dans le code source.

La troisième option consiste à étudier le code source lors de la compilation, dans lequel nous pourrions fournir au compilateur des informations supplémentaires lors de l'exécution, par exemple, ce qui se passera dans le futur avec un tampon. Les compilateurs tentent depuis longtemps de réduire la latence d'accès à la mémoire en insérant des instructions de pré-chargement pour des applications spécifiques et/ou des plates-formes spécifiques. Nous pensons que ce type de travail devrait permettre aux compilateurs de détecter la sensibilité à la latence ou à la bande passante des noyaux et ainsi fournir des conseils de sensibilité aux systèmes d'exécution. Par exemple, les accès en continu/linéaires à des tampons contigus peuvent être détectés et marqués comme sensibles à la bande passante sans que l'utilisateur n'ait besoin de comparer ou de profiler manuellement l'application.

Ensuite, les développeurs doivent préparer les applications et les runtimes pour travailler sur les différents systèmes de mémoire hétérogènes, mais il est clair qu'ils n'ont pas toujours accès à des machines avec des scénarios de mémoire différents. C'est pourquoi les outils nécessaires doivent être fournis afin que les développeurs puissent préparer leurs applications à affronter différents systèmes de mémoire hétérogènes. En fait, les développeurs peuvent facilement travailler avec le système de mémoire hétérogène KNL très connu car la configuration reste toujours la même. Cependant, cela change avec les NVDIMM dans les plates-formes génériques, où le nombre ou les types de mémoires peuvent changer de manière significative. Par conséquent, il est nécessaire de tester le runtime/les applications sur une grande variété de configurations matérielles pour s'assurer que notre logiciel est portable.

Tout d'abord, nous avons abordé les scénarios dont le développeur a besoin pour tester les performances de son application déjà développée sur une mémoire hétérogène spécifique. Et deuxièmement, nous avons abordé différents outils, avec lesquels nous pouvons exposer des configurations de mémoire à des applications qui ne correspondent pas précisément à l'environnement physique, c'est-à-dire que nous présentons des outils qui peuvent être utiles pour émuler des systèmes de mémoire hétérogènes. En fait, cela n'expose pas une véritable performance hétérogène, mais dans ce cas, il s'agit d'exposer le système de mémoire hétérogène comme le ferait un système réel.

Lorsqu'on parle de simulation de performances, on fait référence au fait de simuler le comportement que doit avoir une certaine application lorsqu'elle fait face à un scénario mémoire très éventuellement difficile d'accès pour le développeur. Par exemple, le cas d'un futur système à venir avec un système de mémoire hétérogène différent. Nous avons identifié des options qui nous permettent de modifier les performances des applications. Premièrement, l'accès mémoire non uniforme (NUMA) a été généralisé dans les machines multi-sockets, dans lesquelles chaque socket a un contrôleur mémoire associé. En raison de la distance relative entre les processeurs et les dispositifs de mémoire, l'accès à la mémoire n'est pas uniforme. Depuis la naissance des systèmes NUMA, diverses études ont été menées pour tenter de minimiser l'impact de cette situation. Deuxièmement, la technologie Intel Resource Director (Intel RDT) est un outil qui permet de surveiller l'allocation de cache et de mémoire. Il est exposé en tant qu'interface utilisateur pour le contrôle des ressources par le noyau Linux. Intel RDT est capable de partitionner certaines ressources telles que la hiérarchie du cache et pour notre intérêt particulier la bande passante. Et troisièmement, le piratage de mémoire est une méthode de simulation de performances qui permet de contrôler la quantité de mémoire disponible pour l'application en la co-exécutant avec une application qui lui vole de la bande passante.

D'un autre côté, l'émulation d'un système de mémoire permet aux développeurs de préparer leurs applications pour les systèmes de mémoire de nouvelle génération. L'émulation de ces systèmes consiste à chercher en quelque sorte à exposer des dispositifs de mémoire qui ne sont pas physiquement situés dans les nœuds comme s'ils étaient présents. En émulation, on ne peut pas changer les performances comme dans la section précédente. Son utilisation est principalement dédiée à l'exposition de différents systèmes HMS pour préparer ou adapter des applications pour accéder à différents types de mémoire. C'est-à-dire, tester les heuristiques et les algorithmes développés pour sélectionner la cible appropriée pour chaque tampon. L'émulation peut être effectuée à l'aide de plusieurs options, et nous les avons classées en émulation matérielle, émulation de système d'exploitation et émulation logicielle. L'émulation du HMS via le matériel fait référence à l'utilisation d'un périphérique matériel réel pour imiter la fonction d'un autre HMS. Une autre manière d'émuler des systèmes de mémoire hétérogènes consiste à utiliser des systèmes d'exploitation. Enfin, étant donné que la plupart des environnements d'exécution et des applications HPC peuvent lire la topologie matérielle, l'exposition de différentes topologies modifiera leur comportement. hwloc est souvent utilisé comme couche intermédiaire entre la découverte du matériel et de la topologie dans HPC, nous pouvons donc utiliser hwloc pour « mentir » aux applications.

Enfin, nous nous sommes également penchés sur le problème de la co-programmation. Les nœuds de calcul sont de plus en plus complexes, avec des dizaines de cœurs. La co-ordonnancement permet d'optimiser l'utilisation de ces nœuds là où les applications sont capables d'utiliser des ressources matérielles distinctes pour éviter la sous-utilisation des nœuds disponibles. La co-planification de plusieurs tâches sur de tels nœuds est une stratégie utile pour s'assurer que tous les cœurs sont utilisés dans les centres HPC. Cependant, le partage de nœuds entre plusieurs tâches entraîne également des problèmes tels que des conflits dans le sous-système de mémoire ou une pollution du cache.

Le partitionnement des ressources est un moyen intéressant d'éviter de tels problèmes grâce aux fonctionnalités du système d'exploitation telles que les Cgroups dans le noyau Linux. L'émergence des DIMM à mémoire non volatile apporte de nouvelles stratégies pour la gestion des données dans les applications HPC. En fait, ils prennent en charge plusieurs configurations matérielles et logicielles allant d'énormes capacités volatiles au stockage hautes performances, qui peuvent être utilisées comme tampons pour absorber les pics de stockage, ou pour la récupération après une panne.

Cette thèse a été réalisée à Inria Bordeaux - Sud-Ouest et au LaBRI. Après une présentation de l'état de l'art de l'architecture mémoire, nous avons caractérisé les mémoires par des attributs simples. Nous avons fourni une interface que la bibliothèque hwloc expose aux applications pour comprendre la topologie de la mémoire et allouer des tampons. Ensuite, nous avons proposé une stratégie pour aider les développeurs à adapter leurs applications au bon usage de systèmes de mémoire hétérogènes. Comme l'accès à différentes plates-formes hétérogènes n'est pas toujours possible, nous identifions plusieurs façons de simuler les performances de la mémoire hétérogène et d'émuler différentes topologies de mémoire. Enfin, nous avons construit une stratégie qui facilite le partage de plateformes à mémoire hétérogène et non volatile entre les tâches HPC co-planifiées sur les mêmes nœuds.

Contents

List of Figures

List of Tables

—

Chapter 1

Introduction, Context and Motivations

1.1 Benefits of High-Performance Computing

Today humanity is faced with the vision of making a smarter planet. Where our ability to observe, analyse, understand and solve different social and scientific problems requires to have new levels of efficiency, optimisation and sustainability. Engineers, researchers and scientist seek to solve incredibly complex tasks. E.g., oil and gas exploration [5], simulate molecular dynamics to create new materials [90], forecast climate changes [46], discover new drugs for diseases [84], etc. In fact, during the development of this work, the health crisis caused by COVID 19 impacted the entire world. This disease imposed a great challenge on health systems around the world; causing them to exceed their limits.

High-Performance Computing (HPC) is a key factor to solve numerous advanced scientific problems through models, simulation, and analysis. A single supercomputer can contain tens of thousands of processors. The most productive HPC systems have a very tight combination of hardware and software. Hardware for HPC typically includes high-performance CPUs, memory, storage, and networking components, as well as accelerators for specialised workloads.

The Supercomputer Fugaku shown in Figure 1.1 permitted to accelerate the fight against COVID 19. Since May 2020, it has been used to analyse infection risk using three million node hours for simulating thousands and thousands of droplets moving through the space, including a medley of obstacles and airflows [37]. To the date of writing this manuscript. Fugaku, built by Fujitsu, remained number one in the Top500 list [1] of the fastest supercomputers in the world, where it is still three times faster than the nearest competitor. Fugaku

[1] https://www.top500.org/lists/top500/list/2020/11/

reports 442 PFlops/s (1 PFlops/s = 10^{15} floating operations per second) with 7630848 cores [28]. It is considered a pre-exascale system, and the tendency is to reach exascale systems due to problems that HPC cannot solve quick enough. I.e., that we need faster supercomputers,and for that purpose exascale is needed. To reach that we need to improve everything from hardware to applications.

Heterogeneity is being one of the recent factors that allow having ever faster systems. It permits addressing different application needs by combining, for example, CPU for small/complex computations and GPU for big/simple computations. As seen, it is already in computing, and it has recently been coming to memory.

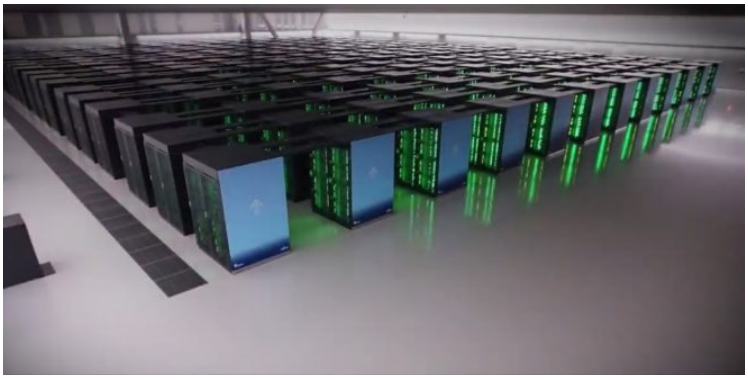


Figure 1.1: The HPC system Fugaku.

1.2 Heterogeneous perspective of HPC

Towards more heterogeneity

HPC is increasingly leveraging heterogeneous architectures. This poses new challenges to better exploit the available resources.

This thesis focuses on the heterogeneity in the memory system. The story begins through the importance of **memory bandwidth**. The speed of many applications is limited by the rate at which data can be delivered from the memory system into the processor. But this factor is not the only one that must be taken into account. HPC applications could use buffers that require memory access with long **latencies**. In the same fashion, very big buffers sometimes also need to be allocate in memory, which challenges the **capacity** of the memory system.

These and many other factors generated a boom of memory devices that seek to improve at least one of these criteria, such as, High Bandwidth Memory (HBM), non-volatile memory, etc. However, these specific types of memory cannot match all criteria at the same time. A high-bandwidth memory is limited in capacity because of price and area constraints. That is why the current trend is for HPC systems to contain heterogeneity in memory systems.

Memory heterogeneity poses many issues

HPC software is originally prepared to support homogeneous memory systems. The way to expose and manage the memory systems considers that there is only one type of memory. The rapidity in the jump to heterogeneity means that HPC software could not adequately support new types of memory or support the fact of having more than one type of memory. I.e., systems have not an appropriate manner to expose or manage heterogeneous memory systems (HMS). The actual support is rudimentary and does not expose the end user to the different characteristics of each type of memory.

The heterogeneity of the memory system makes more complex the development of applications. Applications need to be adapted so that they can properly exploit the memory system and not under-utilise resources. Not all applications have the same behaviour when allocating in a determined memory. I.e., certain buffers of applications have affinities towards some memory kinds.

Another challenge is the fact that applications developers do have not easily access to many kinds of heterogeneous memory hardware systems. It is necessary to provide tools to ease this transition. It means, that developers should be able to predict the behaviour of a certain memory system, and also they should have a manner to emulate heterogeneous memory systems so that the applications can be executed successfully in real heterogeneous systems.

1.3 Outline of the manuscript

This thesis aims to provide tools and strategies to developers to better afford the continuous heterogeneity evolution of memory systems and to have better productivity and portability on the applications.

- Chapter 2 details HPC architectures, the state of the art of the memory subsystem, and state the problems of having heterogeneous memory.
- Chapter 3 presents an interface that takes into account memory attributes to help to expose and manage the memory system complexity.
- Chapter 4 introduces strategies to allow reaching a better criterion about where should buffers have to be allocated.
- Chapter 5 describes to the developers some strategies to simulate the behaviour of a heterogeneous memory system as well as tools that allow the emulation of a heterogeneous memory environment.

- Chapter 6 explains strategies for partitioning non-volatile memory between co-scheduled jobs.

Chapter 2

Memory Systems in HPC

Groundbreaking scientific discoveries are made through data. It leads to innovation and a greater quality of life for thousands of millions of human beings. High-Performance Computing (HPC) is at the base of scientific and social advancements such as self-driving model development, tracking a storm, analysing seismic waves, analysing stock trends, advancing in precision medicine, etc.

Hence, HPC gives the ability to process data and perform calculations at high speed.

Within HPC, the memory system is very important as it helps to store data in memory or in storage. It is presented in a high range of flavours, which allows solving different requirements that an application may have.

This chapter seeks to present the state of the art of memory systems in HPC. First, it presents an overview of the HPC architectures. Then, the memory hierarchy is described, taking into account each of its levels. Later, some emerging memory types that seek to bridge the various gaps in the memory-storage continuum are introduced. Then the state of the art of related software to the use of heterogeneous memory systems is presented. Finally, the main problem that this thesis has sought to solve is propounded.

2.1 HPC Architecture

The effect of having put a group of computers to work on the same task stems from the need to solve complex scientific problems. For this, it was necessary to think that the different calculations should be done in a parallel way so that the times to the solution of a problem can be reduced. Today, the amount of data that must be processed in short periods has increased exponentially, so there is a constant evolution in the conception of High-Performance Computing systems.

In the 70s, supercomputers used only a few processors as was the CRAY era, jumping from 133 MFlops from CRAY-1 [85] (with one processor) in 1976 to Cray-Y-MP [71] with 2.6 GFlops in 1988 supporting multiple processors. An important change in the architecture was observed since 1986 when two or more processors were able to work together using shared memory on Symmetric Multiprocessing (SMP) systems. SMP or Uniform Memory Access (UMA), allows memory operations to be distributed among processors on a common bus as shown in Figure 2.1. However, if two or more CPUs try to access memory concurrently, it could incur bus contention [11]. CRAY-X-MP [23, 94] had a shared memory system where CPU cores can access the same memory simultaneously. But there was a big limitation in this approach, due that all CPU cores having to compete for accessing memory over a shared bus [26]. The conception of these architectures was unique to each vendor, and their main characteristic was to have internal parallelism.

The cost of fabricating faster and faster processors began to increase. In the 90s, machines with thousands of processors appeared [31], each with shared memory, an operating system, and all interconnected by a network. It gave rise to distributed memory systems that are considered the architecture for the construction of modern supercomputers, thus providing a better price-performance benefit. The set of computer nodes is called a HPC cluster and

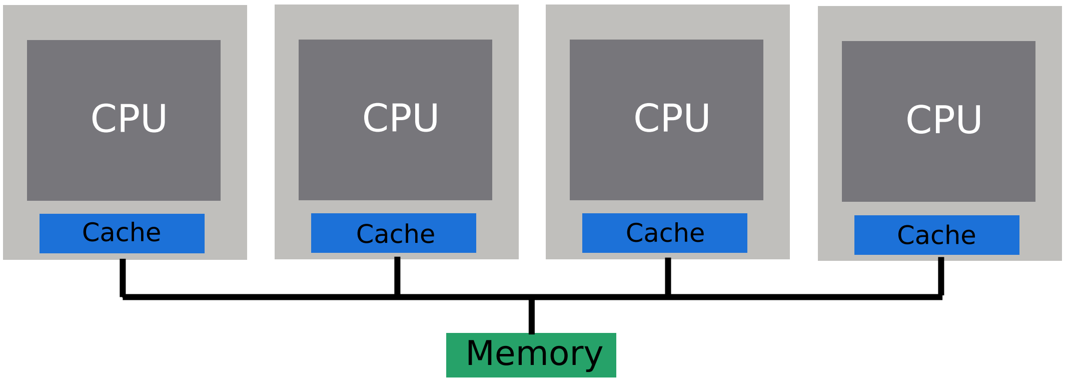


Figure 2.1: SMP architecture.

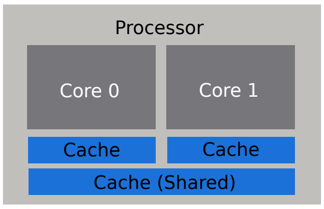


Figure 2.2: Dual-Core processor.

those allow massive parallelism by adding external parallelism. I.e., that instead of multiplying the computational units inside the machine, they could collaborate as individual entities (nodes) communicating over a dedicated network.

HPC clusters are the predominant HPC technology these days. They consist of hundreds or thousands of compute servers that are networked together, to create what appears to end-users as a single highly available system. Each server is called a node, and it works in parallel with the other, improving the processing speed to deliver high-performance computing. HPC clusters in 2005 used single-core processors, often with two processor sockets per cluster node. But due to the increase in the processor clock speed for each new generation, processors faced problems related to the memory speed (due to the increasing gap between processor and memory), instruction-level parallelism (due to having not enough parallelism in a single instruction stream), and power wall (due to the increased operating temperature caused by the high processor frequencies). Given these challenges, the multi-core era began by putting more processors into a single processor substrate as we can see in Figure 2.2. As of 2016, 80% of the systems on the Top500 list have between 6 to 12 cores per processor socket.

Exascale computing is expected to arrive soon. However, conventional approaches as mentioned before are not expected to meet this goal. Co-designed strategies (developing partnership with computer vendors and application sci-

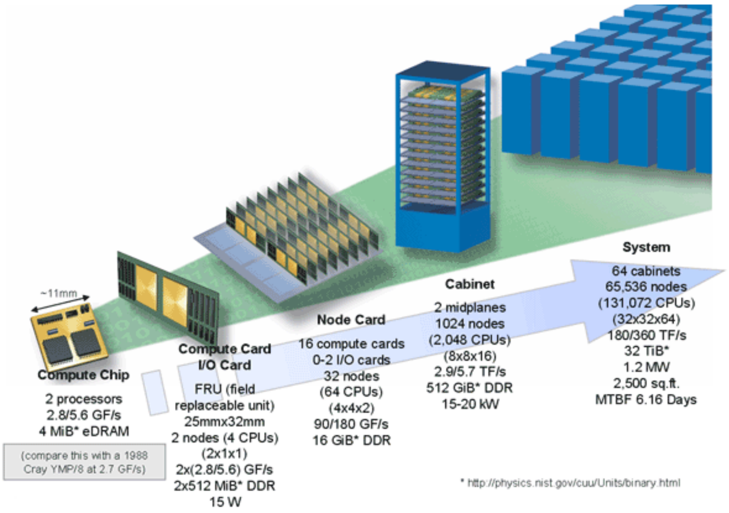
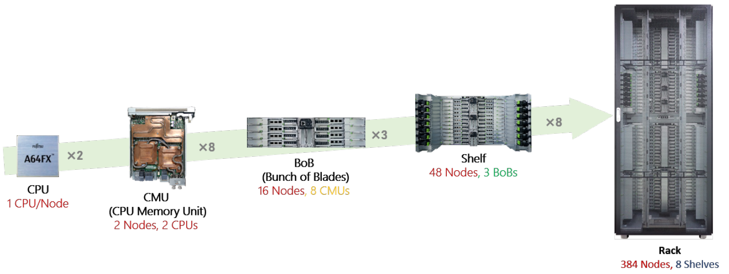


Figure 2.3: Hierarchy of Blue Gene processing units [55].



Courtesy of FUJITSU LIMITED

Figure 2.4: Hierarchy of Fugaku processing units [28].

entists) are being used for hardware and software to meet this performance goal [98, 17]. But this is not a new concept. There were past levels of co-design such as in 2001 when IBM and Lawrence Livermore National Laboratory (LLNL) collaborated to build the Blue Gene/L (BG/L) supercomputer [4] as Figure 2.3 shows. Twenty years later in 2021, a pre-exascale HPC system appeared with the Fugaku supercomputer [88, 51]. As Figure 2.4 shows, the CPU contains A64FX processors with ARM technology that exposes 48 cores. [4]

HPC nodes are high-end computers in terms of power and performance. Those are based on an architecture designed by the mathematician John von Neumann [59]. The von Neumann computer is composed of a **central processing unit (CPU)** that is connected to **memory** by a communication channel, also called bus [32]. Here instructions and data are stored in memory and then moved to and from the CPU across the bus. The speed of the node depends on the time that the CPU takes to execute individual instructions, and also by the generated overhead that involves moving instructions and data between the memory and the CPU.

Despite this, it does not matter how fast the CPU could theoretically be if instructions and data cannot get in or out of memory fast enough. Therefore, the access to memory speed creates a performance bottleneck in the von Neumann architecture. One of the solutions to this consists of using a larger memory hierarchy to improve performance and minimise cost.

We are not limited to only using von Neumann's computer reference components. We may have specific nodes with Graphic Processing Units (GPUs) [75], accelerators, or offloading engine hardware for specialised workloads [1], and different memory solutions which are described in Section 2.2 [67]. However, the focus of this thesis scopes core-memory subsystems.

2.2 Memory Hierarchy

The memory subsystem of modern computers can be explained as a pyramidal organisation of memory levels where three statements are accomplished:

1. The shorter the access time, the higher the cost.

2. The higher the capacity, the lower the cost per bit[1].

3. The higher the capacity, the lower the speed (higher latency).

Therefore, it is sought to have sufficient memory capacity with a speed that serves to satisfy the demand for performance and at a cost that is not excessive. Yet, thanks to the principle of locality, in where applications tend to access the same range of memory locations repetitively over a short period of time, it is feasible to use a mix of the different memory types and achieve performances close to that of faster memory.

Figure 2.5 represents the memory-storage continuum, comprising several layers of memory technologies that are ordered from the fastest and least dense to the slowest and densest [72].

The pyramid is divided into two main sections: the memory side and the storage side. It normally coincides respectively with the memory bus and the i/o bus. In general, it continues to be the case, but with the appearance of the persistent memory layer, we can find ourselves in the situation where a layer is both for memory and storage.

In the following subsections, the pyramid is detailed from its smallest element, the registers, to the largest displayed elements within the storage. It is important to mention that the memory-storage continuum can be represented with different pyramids depending on the approach [50, 74, 89]. In our case, it adapts to the main elements used within the thesis without neglecting important aspects such as the gap between memory and storage.

[1] Cost per bit is the price of a memory device divided by its capacity

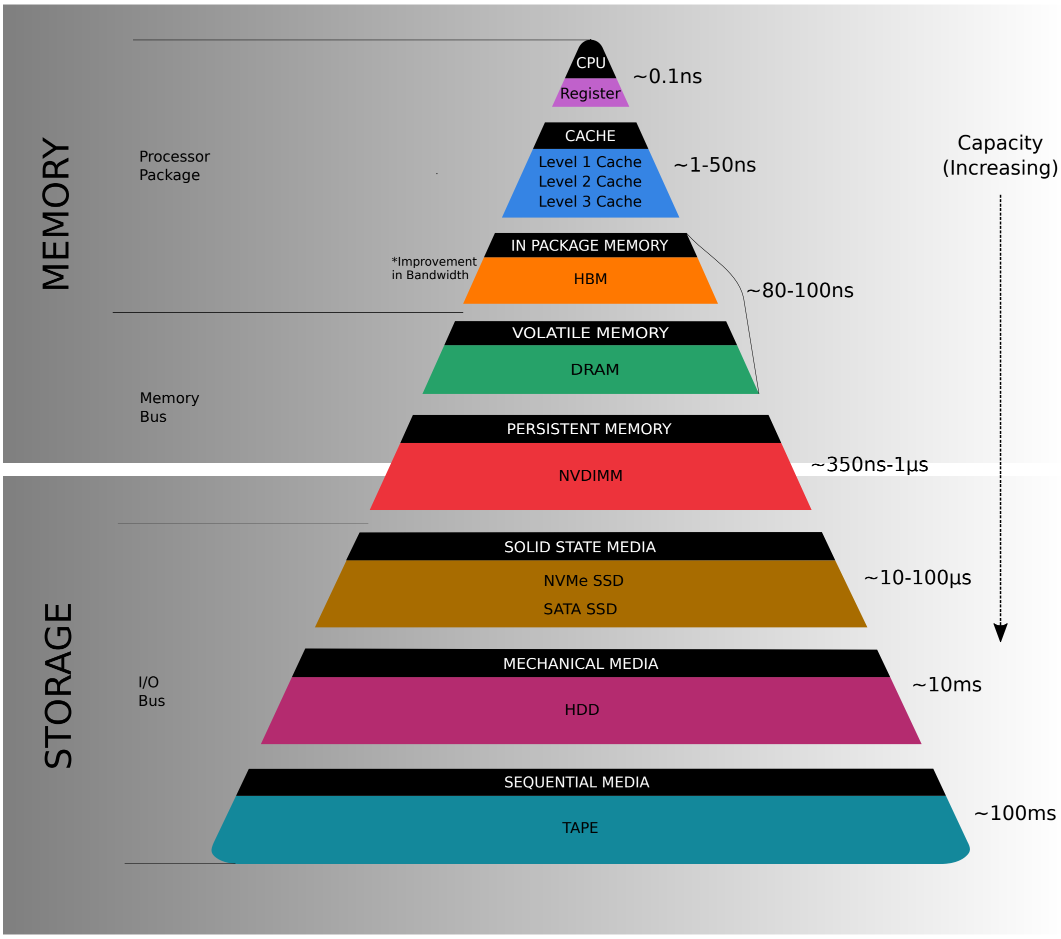


Figure 2.5: Memory-Storage continuum.

2.2.1 Registers

In computer architecture, a register is a high-speed low-capacity memory, integrated into the microprocessor, which allows temporary storage and access to often used values. Registers are at the top of the memory hierarchy and they are the fastest way for the system to store data. Normally they are measured by the number of bits they store; for example, an "8-bit register" [42, 91] or a "32-bit register". Load-store architecture processors used in HPC allow [38]: loading data from the main memory to the registers, performing operations on the registers, and the storage of results in the main memory. They are very limited in number, and therefore it is necessary to take care to optimise their usage to avoid having to go through the main memory or cache again. This concept evolved into Advanced Vector Extension AVX, which is a specific instruction-set designed to work with Intel and AMD into their x86 processors.

Optimising register usage is important to avoid accessing the next memory level. In fact, this proposition is also valid for cache memory hierarchy.

2.2.2 Cache Memory Hierarchy

Cache memory is a very fast memory, which is normally only managed by the hardware. It may be organised as a hierarchy, where the farther away a level is from cores, the higher its capacity and lower performance [69]. Cached data can be hosted at different levels [18] depending on how often it is used. Then, the information can be transferred between the different levels in an inclusive or exclusive way: **the inclusive way** allows the requested data to remain in the provenance cache, that is, a copy is kept at two or more levels; while in **the exclusive way** the requested data is removed from the provenance cache once transferred to the new level [10]. There is a direct impact on cache performance due to the way an application's memory accesses are organised [80].

2.2.3 Main Memory

Main memory corresponds to the hardware where the application's instructions and data are kept when processors are using them, i.e., when the application become active. They are copied from storage into the main memory in where the processor is capable to interact with them. Ideally, in its design, it should have large capacity, low latency, high bandwidth, and low cost. Combining these parameters, hardware designers have given different solutions depending on what is required; meaning to have different technologies that can be used as the main memory device. Most of the main memory modules correspond to volatile memory, which is defined as the computer memory that requires power to maintain the stored data. I.e., it retains information while powered but when power is interrupted, the data is quickly lost.

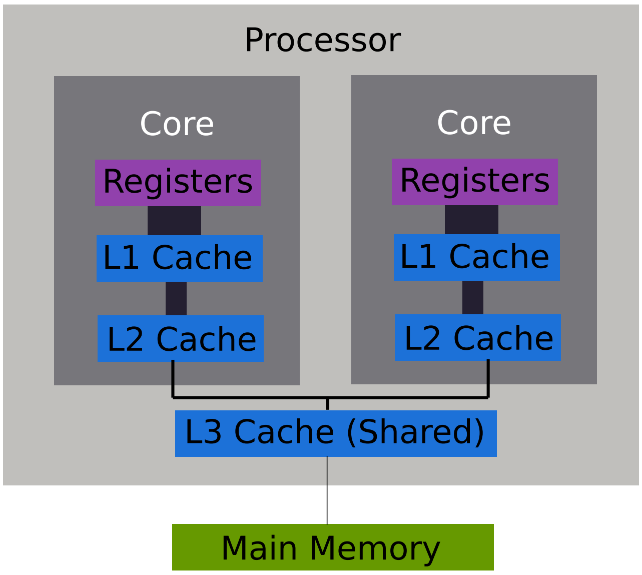


Figure 2.6: General structure of the cache hierarchy.

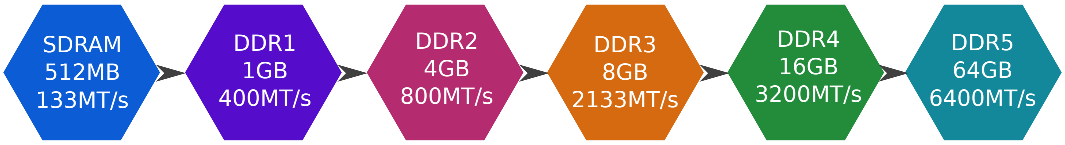


Figure 2.7: DDR evolution in terms of capacity and speed.

Volatile main memory is composed of dynamic random-access memory (DRAM) chips that are typically packaged in dual inline memory modules (DIMMs) [67]. DRAM works synchronously with the system clock, and it is commonly named single-data-rate (SDR) synchronous DRAM (SDRAM). Today, double-data-rate (DDR) SDRAM, an evolution of SDRAM, is used mainly for computer applications in DIMMs. Figure 2.7 shows that today it is possible to find DDR at 64GB with a speed of 6400 million transfers per second (MT/s) compared with SDRAM with 512MB of capacity and 133 MT/s in speed.

2.2.4 NUMA

Non-Uniform Memory Access (NUMA) is a shared memory architecture used today in multiprocessing systems. It was designed to expose different memory nodes. In theory, the access to local memory is without contention, however, it started to appear when large SMP nodes were connected to the memory bus.

NUMA nodes increase the available bandwidth of DRAM [29]. Initially, NUMA nodes were composed of SMP nodes that are interconnected by a communication network that enables the distribution of the memory, as Figure 2.8

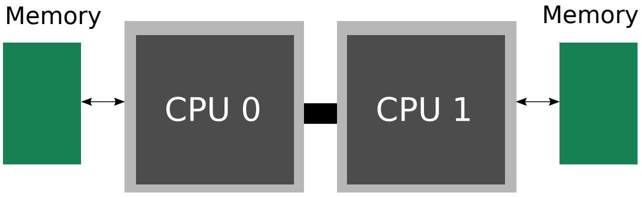


Figure 2.8: NUMA as a set of SMP nodes.

shows.

Access times to memory are relative to the position of the processor and the accessed memory, i.e., if a processor accesses its local memory the access time will be lower than if it accesses remote memory.

In practice, when HPC applications use NUMA, systems can face two difficulties: the first one happens when an HPC application's thread located on a node accesses data located in the memory bank of another node; this non-local transfer hurts performance. The second big issue is due to contention, which happens when two threads located on different nodes access memory in another node (fighting for memory bandwidth).

The use of many cores inside the CPU brings back the contention issue, hence, there are now multiple memory controllers and NUMA nodes inside big CPUs.

To expose these differences within a processor, Intel uses Cluster-on-Die (COD) in Haswell microarchitecture allowing even to subdivide into NUMA domains [2]. Intel Skylake microarchitecture presents NUMA nodes as in Figure 2.9, where three DIMMs per channel are attached to their respective CPUs. Here, is also allowed to subdivide NUMAs with the Sub-NUMA Clustering(SNC) feature, and when it is enabled as in Figure 2.11, the mapping is configured in a way that addresses to half of DDR only maps to the upper region of the processor, whereas addresses the other half only maps to the lower region.

In NUMA nodes the maximum available bandwidth is the sum of the peak bandwidth of each memory, and this can be reached when all cores access their local memory. In Figure 2.10, we are able to see the evolution of bandwidth until using all cores bound to NUMA node 0 in an Intel platform. To maximise the usage of NUMA nodes, developers have to minimise the number of remote accesses by balancing the load between the nodes [47].

[2] https://software.intel.com/content/www/us/en/develop/articles/intel-xeon-processor-scalable-family-technical-overview.html

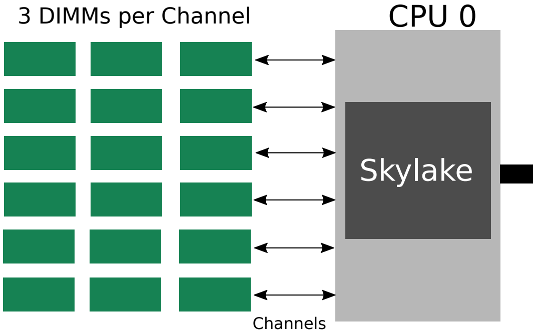


Figure 2.9: Skylake NUMA configuration.

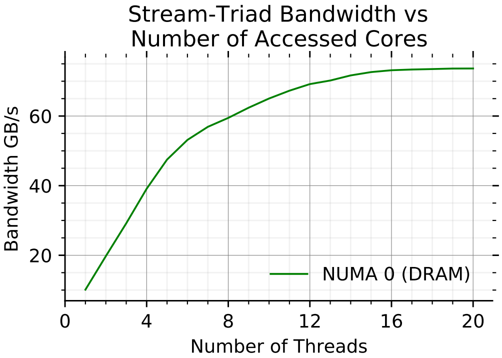


Figure 2.10: STREAM-triad bandwidth peak using a thread per core (20 in total) in NUMA node 0.

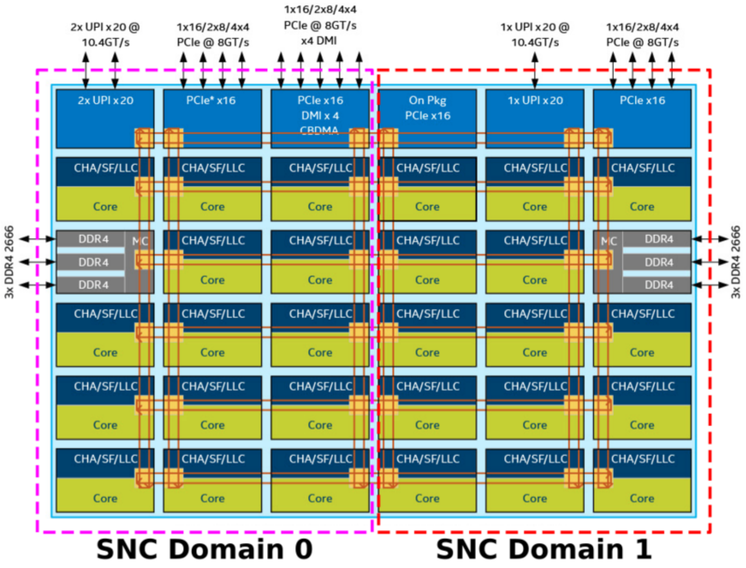


Figure 2.11: Skylake SNC disposition [108].

2.3 New Memory Technologies

Researchers focus their efforts on developing new kinds of memory to fulfil some gaps in the memory continuum primarly related to latency, bandwidth, or capacity issues.

There is still a large gap in the memory continuum between main memory and storage devices that leads vendors to expand their catalogues with different types of memories and even add extra layers as seen in Figure 2.5.

2.3.1 HBM

High Bandwidth Memory (HBM) is a memory device where several DRAM chips and optional IO/controller chips are piled and interconnected by Through Silicon Vias (TSVs) [49]. This kind of memory is very efficient when is placed on a silicon interposer next to the computing chips in the same package [7]. The idea of HBM is to achieve higher bandwidth than DRAM by stacking DRAM dies. Their second generation, or HBM2, specifies up to eight dies per stack. HBM must be connected to the CPU through a specific very wide bus (in comparison to DRAM).

2.3.1.1 KNL

Knights Landing (KNL) is the codename for the second generation of Intel® Xeon Phi processors. As Figure 2.12 shows, this processor contains on-package 3D-stacked memory implemented as a Multi-Channel DRAM (MCDRAM) [78] and unlike DIMMs it cannot be removed or replaced. MCDRAM compared to DRAM, differs significantly in some metrics such as capacity and bandwidth. E.g., in [86] STREAM-triad bandwidth output for DRAM is about 60 GB/s and MCDRAM is about 290 GB/s. MCDRAM is different from HBM as defined by the JEDEC standard. However, we consider them identical since their impact on software is the same.

KNL was created to support a huge amount of full-fledged threads. When using normal main memory, the memory controllers can be overwhelmed rapidly and cause a degradation in performance by the stalling of memory requests and idling threads [83]. HBM narrows this gap by providing high bandwidth to applications.

The KNL memory subsystem can be configured in three different modes: Flat, Cache and Hybrid.

In Flat mode, as Figure 2.13 shows, the user is seeing HBM as a memory pool exposed by the operating system as an extra NUMA node. For this system, NUMA node 0 corresponds to DDR memory and NUMA node 1 is the on-package HBM.

In Cache mode, as Figure 2.14 shows, HBM becomes transparent to the OS and is managed by the hardware as a large cache in front of DRAM. Here,

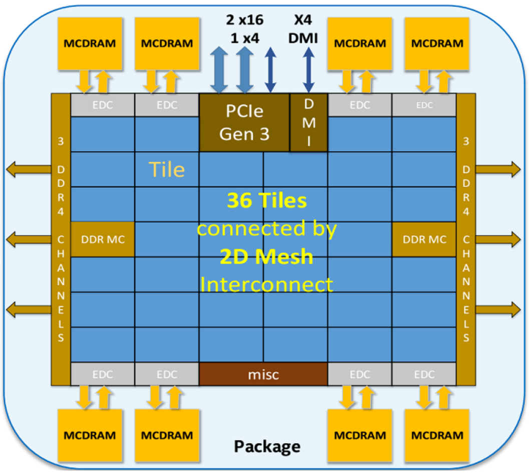


Figure 2.12: Intel Xeon Phi processor overview [40].

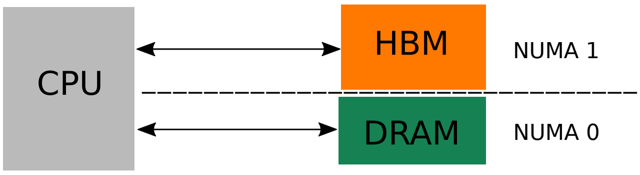


Figure 2.13: HBM configured in Flat mode.

HBM is treated as the last level cache (LLC), which is located between DDR and the L2 cache, into KNL processors. The main advantage of this memory mode is that it is managed by the platform and it is transparent to software, i.e., developers do not need to modify their applications.

Finally, Hybrid mode, as Figure 2.15 shows, allows using parts of HBM in Flat and Cache mode indistinctly and separately.

Developers, due to the fact of having two kinds of memory in these systems, must use them in different manners, depending on the hardware configuration, and have to explicitly manage the two kinds of memory for their applications.

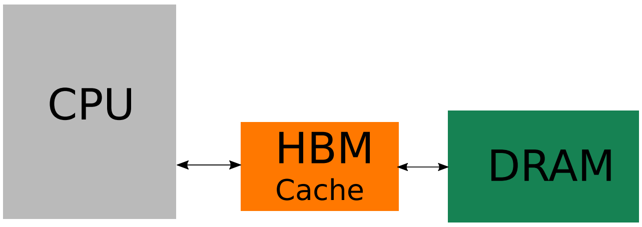


Figure 2.14: HBM configured in Cache mode.

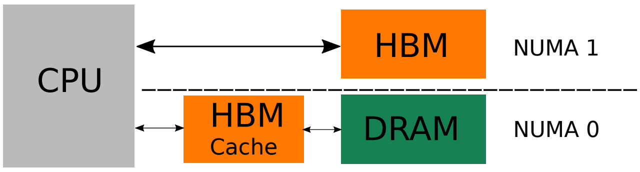


Figure 2.15: HBM configured Hybrid mode.

2.3.1.2 HBM in ARM

The Fugaku system, ranked first in the most recent Top500 (June 2021), is based on an ARM architecture architecture adopting Scalable Vector Extension (SVE) [88]. This system was built with the Fujitsu microprocessor A64FX. Each processor contains 32GB of HBM2 memory with an aggregated bandwidth of 1TB per second.

Into the road to better support artificial intelligence, the ARM architecture and HBM have been integrated, forming a processor named K-AB21 that not only contains HBM2 but also have DDR5 memory, having as a result a heterogeneous memory system.

In the same fashion as K-AB21, Rhea chips (ARM-based) support also HBM and DDR5.

2.3.2 Non-volatile memory

Some technologies have appeared to pursue the expected evolution of Non-Volatile Random-Access Memory (NVRAM) such as Phase Change Memory (PCM), Ferroelectric RAM (FeRAM), Conductive Bridging RAM (CBRAM), etc [109]. NVRAM refers to the memory that can hold data even when power has been turned off. Indeed, new NVRAM technologies do not need to refresh to keep data persistent.

NVRAM can be used either as storage or as memory. However, to understand its nature, it is important to understand the complex problem of the memory-storage gap; where it is evident that there are two sides to the narrative.

2.3.2.1 Storage, slow but persistent

The advantage of NVRAM on disks is obvious: due to their non-volatility nature and the high performance that NVRAM gives over Solid State Drives. Also, NVRAM removes the risk of mechanical failures.

In the pyramid, we can see that each layer from bottom to top makes data more rapidly accessible to the processor. And approximately, it is expected that each layer is 10 times the capacity of the layer above, but one-tenth the performance.

Table 2.1: HDD 1957 vs 2021 comparison.

HDD Property	IBM 350 (1957)	Seagate HAMR (2021)	Change
Maximum Drive Capacity	4MB	20TB	5×10^6x Better
Average Seek Latency	25ms	7ms	2.5x Better

The jump in the storage, as we can see in Table 2.1, shows that there has been a great advance in memory capacity. In the case of the Hard Disk Drive (HDD) in 2021 this has been 5×10^6 times better than its version of 1957. However, if we consider the evolution in terms of latency, it is barely evolving at 2.5 times. It causes to have a storage-performance gap between main memory and storage.

It consequently has put pressure on the default main memory in the nodes, and as expected it brought different technologies that seek to improve HDD performance. To do this, later implementations changed the way to access a physical location of data without the mechanical arm movement. In general, these technologies are grouped as Non-Volatile Random Access Memory (NVRAM) and can be used for both main memory and storage. One of these technologies is the NAND Flash-based Solid State Disk (SSD) which achieves much faster random access speed than the traditional HDDs (up to 100x) [43]. This technology has stalled, and it only evolves in terms of capacity, doubling every 2 years, but maintaining almost constant latency. In addition to this, CPUs are getting faster and the storage-performance gap remains [70].

2.3.2.2 Memory, fast but volatile

The other side of the story starts with the evolution of the capacity of main memory devices that doubles every 4 years approximately compared to data sets doubling every 3 years; meaning that applications cannot allocate as much data as they would like close to the processor; causing us a memory-capacity gap.

Non-volatile DIMMs (NVDIMMs), are the NVRAM that we put on a memory module (DIMM). With the emergence of NVDIMMs, the gap between usual volatile memory (≈80ns) and persistent storage (≈50μs to 7ms depending on the technology) is expected to narrow to avoid as much as possible wasting CPU cycles waiting for data. It is important to consider that this gap gets bigger considering that CPU continuously increase its performance, making memory-storage data further away from it; this is most acutely felt in data centres where data sets get 2 twice as big approximately every 3 years.

NVDIMMs bring the ability to HPC systems to have a high-capacity slower memory, i.e., they provide a very high-capacity compared to DRAM, but at the cost of higher latency.

2.3.2.3 NVDIMM implementations

There are three very different standardised NVDIMM configurations according to JEDEC (who also standardises DIMMs for the DRAM). In Table 2.2 we can observe NVDIMM-N, NVDIMM-F and NVDIMM-P.

NVDIMM-N uses DRAM and NAND Flash in the same module. DRAM is accessed directly and NAND Flash is only used for backup. In the event of a power failure, DRAM data is copied to flash memory, and copied back when the power is restored. To do this it has a small power backup in the form of a battery. It is constrained by DRAM capacity.

NVDIMM-F is essentially an SSD that uses the DDR3 or DDR4 bus instead of Serial Advanced Technology Attachment (SATA), Serial Attached SCSI (SAS) or Peripheral Component Interconect Express/Non-volatile Memory Express (PCIe/NVMe). Although it provides lower latency than normal SSDs, it essentially works the same as SSD. That is why it is at the level of tens of microseconds and the capacity can reach terabytes.

NVDIMM-P, fully released in 2021, enables persistent memory technologies over DDR4 and is prepared to work with DDR5, meaning that it could benefit from its bandwidth improvements. In essence, NVDIMM-P combines DRAM and non-volatile memory, supporting Byte or Block access protocols.

For the thesis interest, it is important to mention that the only device available was the Intel Optane DC persistent memory (in section 2.3.3.2). Technically, it does not implement the NVDIMM-P standard, but it is similar conceptually.

Table 2.2: NVDIMM types.

	NVDIMM-N	NVDIMM-F	NVDIMM-P
Access Method	Byte or Block	Block	Byte or Block
Capacity Range	DRAM (tens of GB)	Flash (TB)	NVM (TB)
Latency	DRAM (tens of ns)	Flash (tens of μs)	NVM (hundreds of ns)
Other	Requires battery or capacitor	Flash SSD on a DRAM bus	Multiple media types

2.3.3 Intel non-volatile memory solutions

Intel has a very important role in the development of NVRAM, allowing a significant advance in the development of this new memory layer. It permits having different devices that can connect either to PCI, SATA or memory DIMMs. The latest Intel technology developed with Micron in 2012 is the 3D XPoint under the brand name Optane. It is transistor-less, bit-addressable, faster and durable compared to NAND technology. It can act as memory (byte-addressable) but also as traditional storage (block addressable) [19].

This thesis focused on Intel Optane 3D Xpoint, because it is the only widely available product. For this reason, it is important to take into account that future mentions of NVDIMMs in the following chapters refer to Intel Optane 3D Xpoint technology.

2.3.3.1 Optane SSD

Intel Optane technology is used in SSDs (Intel Optane DC SSD). It complements Intel QLC 3D NAND SSDs providing higher input/output operations per second (IOPS), low latency and can better afford heavy write workloads. If we compare both technologies the main difference is that Intel Optane DC SSDs are better to address input/output (I/O) and Intel QLC 3D NAND SSDs are better for large-capacity storage. They could be used in conjunction and Intel Optane's version could be used as a fast-caching storage layer for hot data in front of the 3D NAND version [19].

2.3.3.2 Optane DCPMM

Intel has put in the market the Optane Data Center Persistent Memory Module(DCPMM) as its last NVDIMM solution providing byte-addressability and a higher capacity than DRAM and compared to the SSD version it provides lower latency and higher bandwidth. Unlike DRAM, they can retain data across power cycles. The main idea of these DIMMs is that they are inserted in the usual slots just like DDR DIMMs as Figure 2.16; even do they co-exist with conventional DDR4 and DDR5 DRAM DIMMs on the same platform since the 2nd Generation of Intel Xeon Scalable processors.

They are configured as individual Regions or as Interleaved Regions. The main advantage of using interleaved regions is that they increase the memory bandwidth because it uses multiple channels simultaneously. However, if something fails in one NVDIMM, the entire region data is lost.

Besides regions, the lasts Xeon processors Ice Lake and Cascade Lake are capable to use these NVDIMMs as normal (volatile) memory. For this, NVDIMMs have two operating modes:

1. The 1-Level-Memory mode allows using the NVDIMM as both persistent storage as Figure 2.17 and normal memory as Figure 2.18.

2. The 2-Level-Memory mode only allows using the NVDIMM as normal memory having DDR as cache in front, as Figure 2.19 shows.

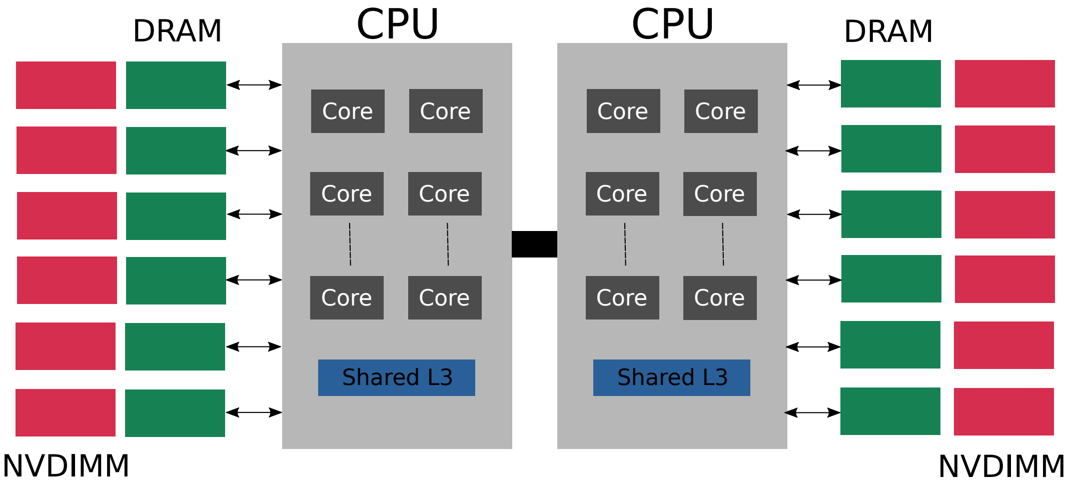


Figure 2.16: Dual-socket Xeon platform with 6 channels per processor, with one NVDIMM and one DDR each.

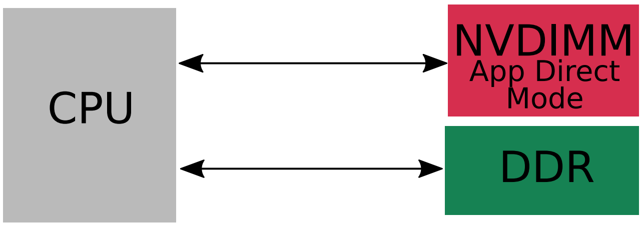


Figure 2.17: 1-Level-Memory mode (1LM) using App Direct mode, using DDR as the main memory while NVDIMMs are exposed as a persistent memory region that is usually used as storage.

The configuration is done within the BIOS, or by a tool named ipmctl to partition them between the target modes. **It is important to mention that the following chapters focus on using the NVDIMMs in 1-Level-Memory mode as normal memory**. All other modes are explained in Chapter 6.

2.3.4 Other memories

There are more kinds of memories that are already available on the market. For instance, NVIDIA's V100 GPUs expose their internal HBM as additional NUMA nodes on POWER9 processors such as Figure 2.20 shows. In addition, some NVMe drives can expose some regions as NVDIMMs.

The actual manner that HBM, DRAM and NVDIMM are connected to CPUs is fixed in hardware. DRAM and Intel NVDIMMs use external DDR slots, while HBM is embedded inside the processor package. HBM has a limit of stacking that basically limite their capacity. It should be added the fact that, unlike DRAM and NVDIMM, HBM are not field-upgradable. DRAM in

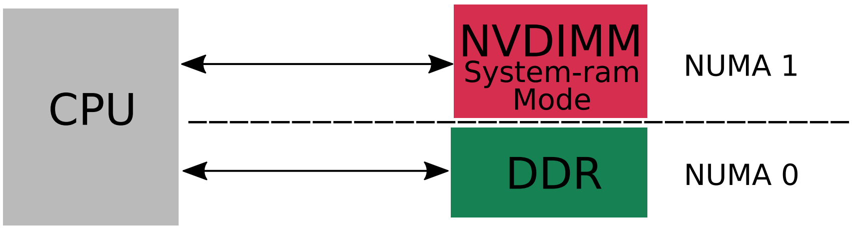


Figure 2.18: 1-Level-Memory mode (1LM) using System-RAM mode, allowing to appear the NVDIMM as an additional NUMA node apart of DDR.

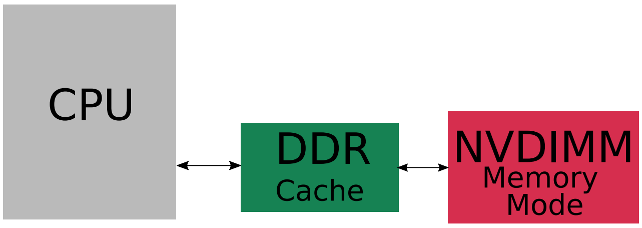


Figure 2.19: 2-Level-Memory mode (2LM), using DDR as a Memory-side Cache in front of the Memory Mode part of NVDIMMs exposed as normal volatile memory.

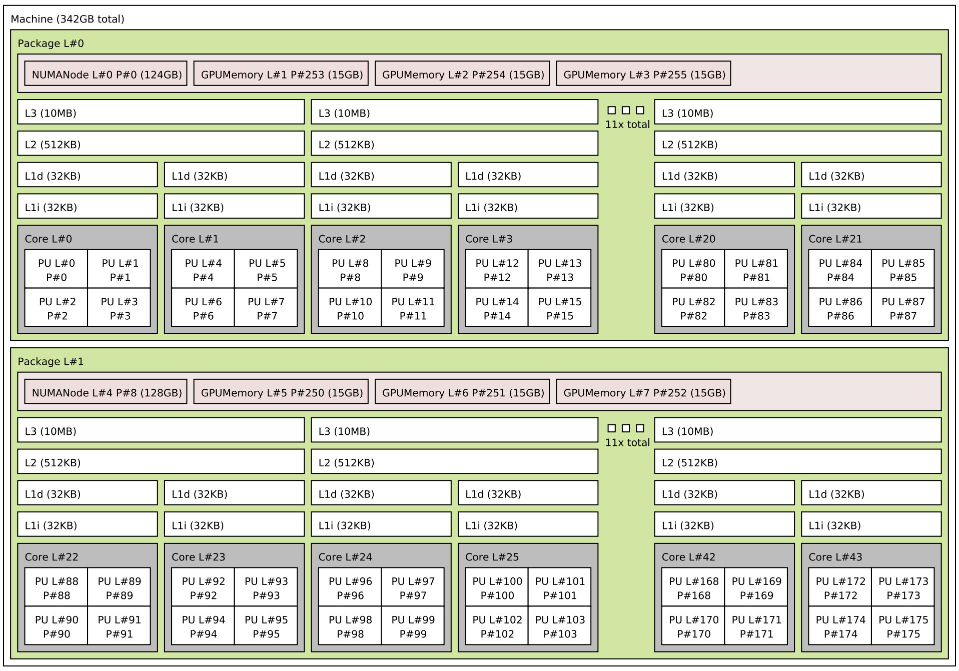


Figure 2.20: hwloc's output with NUMA nodes (pink-colored) where the memory of 6 NVIDIA GPUs are exposed as a NUMA node.

its last forms (DDR4 and DDR5) suffers from a lack of bandwidth and capacity

For this reason, there are multiple proposals to reinvent the way the memory is connected to the CPU. In the POWER10 architecture, the Open Coherent Accelerator Processor Interface (OpenCAPI) enables a new way, dubbed Open Memory Interface (OMI), to connect DRAM and HBM without incurring high cost and capacity restrictions. Indeed, the OMI serial bus [34] allows getting memory closer to the CPU than either DDR slots or HBM embedded schemas.

There are many other proposed technologies dedicated to improving the interconnections inside servers: the Computer Express Link (CXL), is a bus protocol that runs across PCIe 5.0 link and it is supported by AMD, ARM, IBM and Intel architectures[3]; the Cache Coherent Interconnect for Accelerators (CCIX)'s approach to peer-to-peer connections on the bus, taking memory from different devices, to then pool it together and map it into a single NUMA.

Gen-Z allows users of disaggregated data centres to request memory capacity on demand by connecting memory to CPUs between nodes. It promises to maintain good bandwidth and latency through a new dedicated physical interconnect. Network-attached memory [3] is a slower alternative with rather regular hardware: memory in another node or a specific node is made available remotely across a standard network. In both cases, depending on the interconnect performance, this remote memory may be considered local to some CPUs (close to the network interface), or remote to all CPUs (if the network is far slower than usual memory interconnects, as in Figure 2.22).

2.4 Impact of the Memory Subsystem

An ideal memory system should provide the highest bandwidth, the lowest latencies, big capacities and minimum power consumption [68]. However, this is just an idealisation that can not be translated into a memory module.

2.4.1 Combining different kinds of memory

To describe the effect of combining different types of memory, it is necessary to first talk about the effect of not doing it. Fugaku presented in Section 2.1 is the last generation supercomputer where its memory system is homogeneous. Being more specific, it only works with HBM memory. Homogeneous memory systems are normally optimised to provide either low latency or high bandwidth. However, due to the diversity of workloads (in terms of memory access), homogeneous memory systems are not sufficient. I.e., they do not support the diversity of computation- and memory-intensive workloads.

[3] https://blocksandfiles.com/2020/04/03/cxl-gen-z-bus-standards-agreement/

Thus, the effect of combining different types of memory to form a heterogeneous memory system implies mixing memory devices with distinct performance and characteristics. These systems give support to the diversity of workloads. E.g., Intel's Knigth Landings processors mixing HBM and DRAM allow to better support bandwidth- and latency-intense applications. Similarly, Intel Xeon Cascade Lake/Ice Lake with DRAM and NVDIMMs better support latency- and capacity-intense applications. In addition, heterogeneous memory systems are also being used as a technology for last generation platforms such as presented in 2.3.1.2 with K-AB21 and Rhea that mixes HBM and DRAM to provide better support in some specific types of application.

The impact to developers when having heterogeneous memory systems falls on the fact of requiring a more portable, productive and efficient memory management scheme.

2.4.2 Locality vs Heterogeneity

Through memory slots, NVDIMMs are attached to processors. Their access performance suffers from the same locality issues as normal DDR memory, i.e., accessing an NVDIMM is faster from the CPU where it is attached. NUMA effects are not negligible [54], meaning that applications must take locality into account when choosing an NVDIMM target.

In Figure 2.21, we observe that the heterogeneity of the memory system opens the possibility of having different kinds of memory local to a CPU. And in the same fashion to the ones that are remotes, the possibility of having different kinds of memory remote to a CPU. Indeed values of bandwidth and latency can also be retrieved to correspond to a locality such as in Table 2.3.

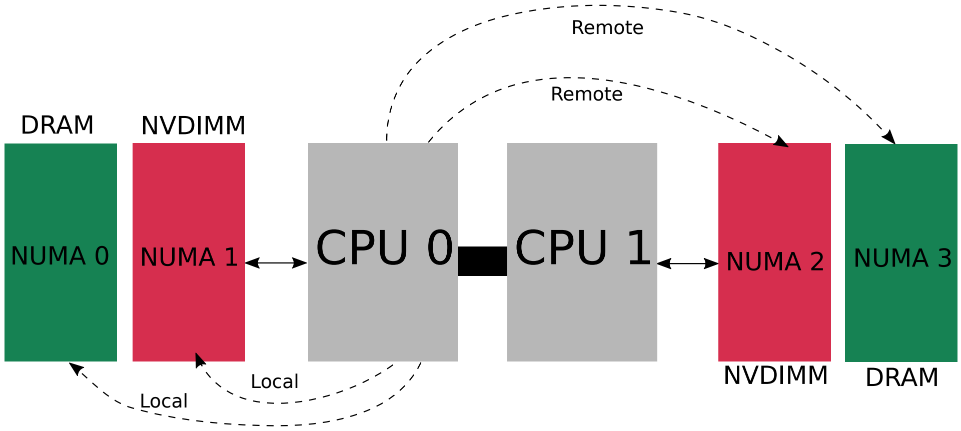


Figure 2.21: Locality of a heterogeneous memory system containing DRAM and NVDIMMs relative to CPU 0.

Here, we can infer a situation when local DRAM is full. The alternatives are to use either Local NVDIMM or Remote DRAM. If we consider just locality, the obvious answer would be choosing Local NVDIMM. However, we must consider that the memory system is no longer homogeneous in terms of memory characteristics. This new heterogeneity, makes us rethink our criteria. E.g., if we take a good look at Table 2.3 we can see that the Local NVDIMM has in terms of latency a similar value to the Remote DRAM, so the decision could fall on still using the Local NVDIMM. However, this does not happen if our criteria are based on bandwidth, as we can see that it would be a better option to use the Remote DRAM.

Table 2.3: Bandwidth and latency values of local and remote targets relative to CPU considering Figure 2.21.

	Target			
CPU 0	0 Local DRAM	1 Remote DRAM	2 Local NVDIMM	3 Remote NVDIMM
Latency ns	84.2	145.5	149.7	349.6
Bandwidth GB/s	81.52	37.14	8.50	2.12

2.4.3 Summary

The memory hierarchy that today can be found in HPC clusters has been growing as processors evolve. Since the data in memory must be processed in an increasingly efficient way. There are gaps that exist within the memory hierarchy. Those have made technology builders not stop optimising existing products. But also, this has generated that the variety of memory types increases, even incurring in new memory types, which are memory technologies that try to fulfil some gaps in the memory hierarchy.

In this way, it is inevitable to have more complex memory systems within an HPC cluster, that is, more than one type of memory can be used by the CPUs on a node. These heterogeneous memory systems, while helping to fill the gaps and bringing data closer to processing, add complexity to the development of applications, libraries, and even how an operating system should manage them. In Figure 2.22 we can observe an example of this by having a system with four kinds of memory.

In Table 2.4 we can observe 3 different platforms (more details in Chapter 3) that contain different kinds of memory inside them. Platform 1 contains DRAM and NVDIMM memory, platform 2 contains HBM and DRAM memory and platform 3 contains HBM, DRAM and NVDIMM memory. The OS allows to see one NUMA node per memory kind. That way applications could allocate wherever they want as if they were in a non-heterogeneous NUMA machine. I.e., CPUs have more than one local memory. That poses in the table the possibility of better supporting different kinds of applications such as bandwidth-, latency- and capacity-bound applications that could benefit from

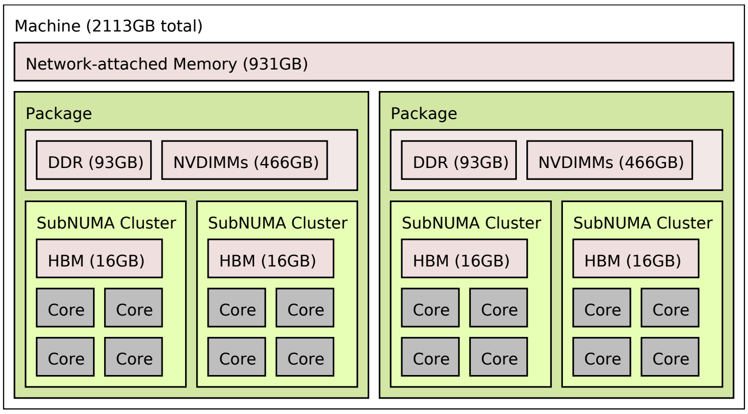


Figure 2.22: Output of hwloc's lstopo tool on a fictitious platform with several kinds of memory: each CPU package has local NVDIMM and DDR NUMA nodes. Each SubNUMA cluster in those packages also has an HBM. And a network-attached memory is also connected to the entire machine.

allocating their buffers to a more suitable memory kind. E.g., bandwidth-bound applications would prefer to use DRAM memory on platform 1, HBM memory on platform 2, and HBM memory on platform 3.

Table 2.4: Comparison of 3 different heterogeneous memory systems with different application needs.

	Platforme 1	Platforme 2	Platforme 3
Default	DRAM	DRAM	DRAM
Latency-Bound	DRAM	DRAM	DRAM
Bandwidth-Bound	DRAM	HBM	HBM
Capacity-Bound	NVDIMM	DRAM	NVDIMM

2.5 Software State of the art

Traditionally, NUMA memory management is done by two main policies: the NUMA interleaving policy, where pages are interleaved evenly on DRAM [24], and first-touch policy, where a page is mapped to the NUMA domain of the thread which first touches it. In addition, there is another policy, the Automatic NUMA balancing policy that aims to migrate data on demand to the memory nodes local to CPUs. However, it is not widely implemented in the operating systems. Having different kinds of memory complicates these policies. It is not clear whether interleaving between different kinds of memory is a good idea (probably having a different performance for the same buffer allocation), because it has not been designed for that. However, there is a proposal called top-tier memory management that aims to give support by migrating infrequently used pages to slow memory and hot ones to fast memory [58].

2.5.1 Managing Heterogeneous Memory

There are several APIs that allow managing more than one memory kind. One of these is memkind library [16] which is an open-source designed to manage HBM. This allows defining memory kinds and permits allocation either in DRAM or HBM. Also, it has an automatic allocator autohbw which do allocations based on a certain size range to HBM at runtime and avoiding source code modifications by intercepting allocations. However, memkind first implementation was not generalised to work in every heterogeneous memory system. Their developers did not try to become generic to heterogeneous memory systems until just recently. Memkind does not support NUMA, which means that it does not take into account the locality of, e.g. HBM, DRAM or NVDIMM. This makes memkind useful in simple problems supporting the allocation process, but not in more complex problems where it is important to decide before allocating.

Memory Object Classification and Allocation (MOCA) [68] is a framework for heterogeneous memory systems that focuses their efforts on memory objects within applications, where, depending on the memory access behaviour, allocations are made in different memory types. For this, it requires doing previous profiling of the application.

Simplified Interface to Complex Memory (SICM) [74] claims to be a universal interface for discovering and managing heterogeneous memory systems supporting HBM, NVDIMMs and DDR. It is composed of tiers: the low level is the one charged to control the use of what kind of memory is being used and provides tools to discover, allocate and de-allocate memory buffers. This high-level provides developers with a manner to specify parameters that could be taken into account in each allocation. SICM requires Linux kernel changes that do not follow the current trend of Linux kernel developers, which is to expose performance attributes to applications that help drive allocation behaviour. Also, the ordering is configured machine-wide in the kernel. All allocation of all jobs running simultaneously on a node will use the same ordering.

Umpire [14, 13] is an application-focused API for memory management on NUMA and GPU architectures. It provides high-performance strategies for customising data allocations. It determines the best way to allocate buffers through the available nodes and resources avoiding that users have to do it manually. It is intended for high-performance applications and provides many memory operations, dynamic memory pools, and introspection capabilities.

AML [79] is a heterogeneous memory management library that provides flexible interfaces to describe how applications deal with data, tiling of data and data placement across different topologies [22]. Then, it permits identify affinities between work and data and it supports DRAM, GPU memory and NVDIMMs.

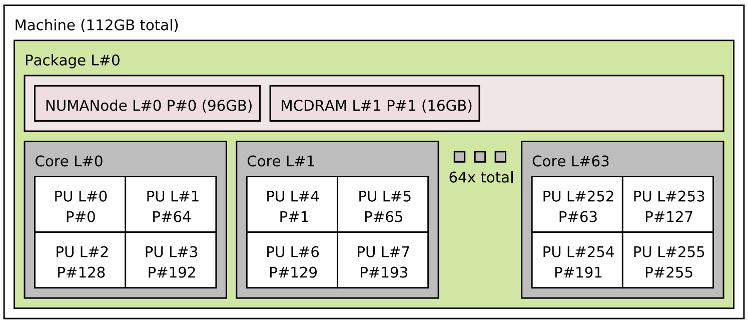


Figure 2.23: hwloc's output on an Intel KNL Xeon Phi platform configured in Flat mode. The CPU exposes DDR memory as a NUMA node and HBM (MCDRAM) memory is shown as an extra NUMA node.

All these approaches have sought to manage heterogeneous memory systems based on simple approaches such as only taking into account the size of a buffer, as well as making complex decisions before allocating. The level of abstraction of many of these solutions often involves a strenuous rework for the developer. Moreover, many of these solutions are limited to a specific heterogeneous memory system. For all this, it is important to have memory management that can take productivity into account, capable of exposing any heterogeneous memory system to applications, and capable of making decisions to allocate the right buffers in the right target.

2.5.2 hwloc

hwloc mentioned in Section 2.2.4 is the de facto interface for exposing locality of hardware [15, 33]. As mentioned before, NUMA nodes were conceived from the relationship between cores and local DRAM memory. With the appearance of new types of memory as seen in Section 2.3, we can find architectures that allow to have two or more different types of local memory for a group of cores. In other words, a group of cores can belong to more than one NUMA node. Before this thesis, hwloc was not able to manage heterogeneous memory systems.

In Figure 2.22, a fictitious machine is shown, containing four types of memory available, where a core has 4 local NUMA nodes. At the package level, DRAM memory is exposed before NVDIMMs due to DRAM normally being the default allocation node. However, this choice does not match the need of every application. In the same fashion, HBM is exposed at the smaller level of this hierarchy and implicitly one expects it to give a higher performance. But, there is no manner to expose that performance information and even worse know the dimension, e.g. latency and bandwidth. A real example could be seen in Figure 2.23 showing the hwloc's output of a KNL system, which contains DRAM and HBM memories attached to the CPU allowing to have two NUMA nodes, where applications could allocate their buffers. This situation adds complexity to how the heterogeneous memory should be used.

On the application side, buffers by default are allocated on DRAM memory regardless of the platform, because the OS and hwloc expose them first. In that sense, NUMA node elements have changed from containing CPUs and memory to have CPUs and several local memories. For that reason, it is better to talk about a relation between an initiator and a target, in where initiators are either a set of logical processors (CPU-sets) or specific objects, and targets are hwloc objects of type NUMA node.

hwloc allows to manage heterogeneous memory systems in a generalised way (i.e., constantly adding support for different HMS). However, before the development of this thesis, it could not expose memory characteristics to help end-users choose the correct target.

2.6 Statement of the Problem

Heterogeneous memory systems have an important role to allow more efficient or faster access to the data in memory from the processors. However, these add a level of complexity to each of the different actors involved in heterogeneous memory system management.

The first level of complexity that heterogeneous memory systems bring is related to how to expose them to applications. First, the need to identify what is inside of the HMS. This is not as easy as one can expect, because we can have different kinds and counts of memory nodes involved and that can not be easily guessed from the OS. Besides this, it is necessary that each identified memory can be characterised and represented in a way that can be expose to the other actors such as OS, applications and developers. **In Chapter 3 we present an extended interface for hwloc taking into account memory attributes that helps to expose and manage the memory system complexity**.

On the other hand, applications also have to deal with the growing number of memory types. Even if the heterogeneous memory system is adequately exposed to applications, it does not mean that they know how to use them or that this process will be automatic. Not all applications will keep their behaviour whether they decide, for example, to allocate their buffers in the DRAM or in the HBM. Therefore, it is necessary to generalise a method to be able to identify beforehand the affinities that an application may have. For this, it is necessary to recognise which are the main buffers or memory objects of an application, that is, the ones for which the memory location has the most impact on performance. **In Chapter 4 are presented some strategies that allow to reach in a better criterion about where to allocate memory buffers**.

Heterogeneous memory systems have various tastes and flavours. It is not always easy to have access to this diversity. Developers need tools to be able to simulate how their applications could behave given an HMS. It is also necessary to be able to emulate these scenarios and be able to present them to future applications. **Chapter 5 collects some strategies and techniques that allow to give developers an idea of how to approach HMS that are not at hand by using simulation and emulation approaches**.

As known computing nodes are increasing with tens of cores. Co-scheduling multiple jobs on such nodes have been a useful strategy for ensuring the use of cores in HPC systems. However, this strategy considers using normally one kind of memory, e.g., DRAM, HBM or NVDIMMs. For this reason, there is a necessity to explore the possibilities to partition and use new memory technologies and handle them with resource partitioning. **In Chapter 6 we provide a strategy that involves the opportunities of partitioning Intel NVDIMMs between co-scheduled jobs on HPC nodes**.

Chapter 3

Navigating Complex Memory Spaces

Managing memory in HPC applications is getting more and more difficult due to the actual heterogeneity of the memory system. To better support these new memory systems, identifying memory kinds and exposing their characteristics is crucial, so that upper software layers could have an idea of where to allocate performance-critical buffers.

Heterogeneous memory systems are not uniquely shaped, and vary from platform to platform, always maintaining an interaction of at least 2 different memory actors or as we called 2-memory-kind (2MK) HMS nodes. In fact, 2MK is the current state of the art in hardware, having combinations of HBM+DRAM and DRAM+NVDIMM, as explained in the following sections. This leads to the need of exposing heterogeneous memory systems to applications in such a way that they can access the different memory types knowing their characteristics and thus give them the best possible use.

In our study, we have mainly worked with 3 different memory environments. The first heterogeneous memory systems used is Kona01. On Appendix A.1 there is detailed information of Kona01 and other Kona machines with similar structure. Figure 3.1 is a simplified representation of Kona01 where we can see that the CPU has an HBM on-package memory module in Flat mode with a capacity of 16 GB and also DDR memory of 96GB. Inside, there are 64 cores of the CPU that has attached 2 different memory kinds, having as a result a total of 2 NUMA nodes.

The second heterogeneous memory system is into Leonide machine (more details in Appendix A.2). Figure 3.2 is a simplified representation of Leonide, where we can see that each CPU has a local DDR and NVDIMM memory. I.e., the 20 cores of each CPU, are attached two different kinds of memory, having as a result a total of 4 NUMA nodes.

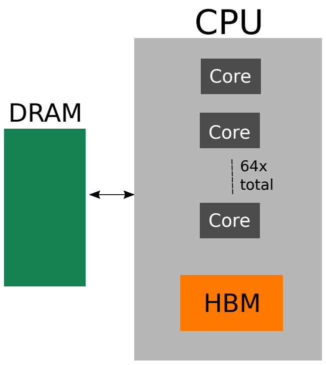


Figure 3.1: Kona01 memory system organisation.

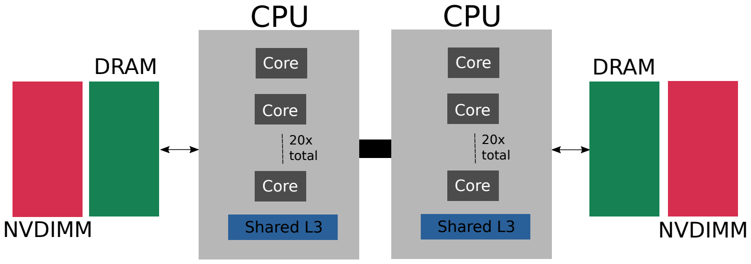


Figure 3.2: Leonide memory system organisation.

Nowadays, to our knowledge, there is still not HPC nodes containing three or more memory kinds. However, in near future, we consider we could find such kinds of HPC nodes. Taking that into account and for explanation purposes, we have imagined a fictitious 3-memory-kind (3MK) HMS node. This heterogeneous memory system consists of HBM, DDR and NVDIMMs. Figure 3.3 is a simple representation of our illustrative machine where the cores of each CPU will have three types of memory as locals, resulting in a total number of 6 NUMA nodes.

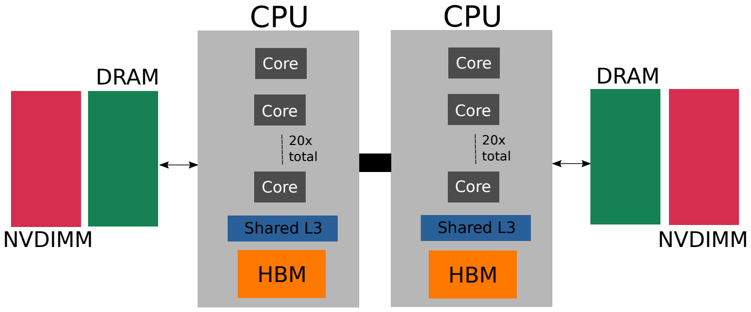


Figure 3.3: 3-memory-kind machine organisation.

This chapter seeks to present our approach to expose complex memory spaces. Considering that the actual manner to expose homogeneous memory systems (1MK systems) is not enough to fully expose heterogeneous memory systems, first section presents a more generic manner to do it. Then, we introduce a set of attributes that can be used to characterise each kind of memory in a heterogeneous memory system. Afterwards, an API that allows to expose and manipulate memory attributes is presented. Then, is presented in detail how can the values of memory attributes are retrieved.

3.1 Exposing Memory Characteristics

Into each heterogeneous memory system presented we have the possibility to observe its topology exposed by the OS and hwloc as shown in Figures 2.23, 3.4 and 3.5.

The memory system is a key component in the topology awareness [33] for have a proper allocation of memory buffers. Operating systems already expose some information to user-space applications, especially in terms of locality. However, Operating systems have not had a complete evolution to support new memory architecture trends. In the same direction, hwloc has a mechanism that allows exposing the memory system of nodes to the userspace. In the past, the hwloc model would place NUMA nodes inside the CPU hierarchy. However, for 2MK or future 3MK HMS nodes, as we can see in Figures 3.4, 2.23, and 3.5 hwloc changed the hierarchical structure to allow

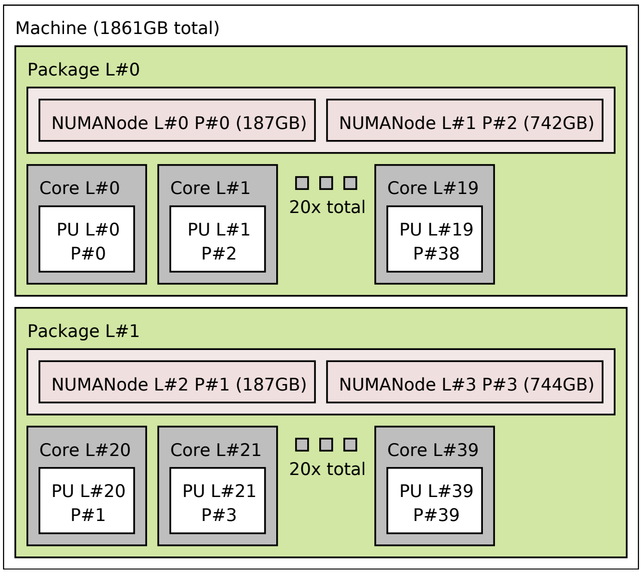


Figure 3.4: hwloc's output of Leonide machine in System-RAM mode. The CPU exposes DDR memory as NUMA nodes and NVDIMM memory is shown as extra NUMA nodes (cache hierarchy hidden).

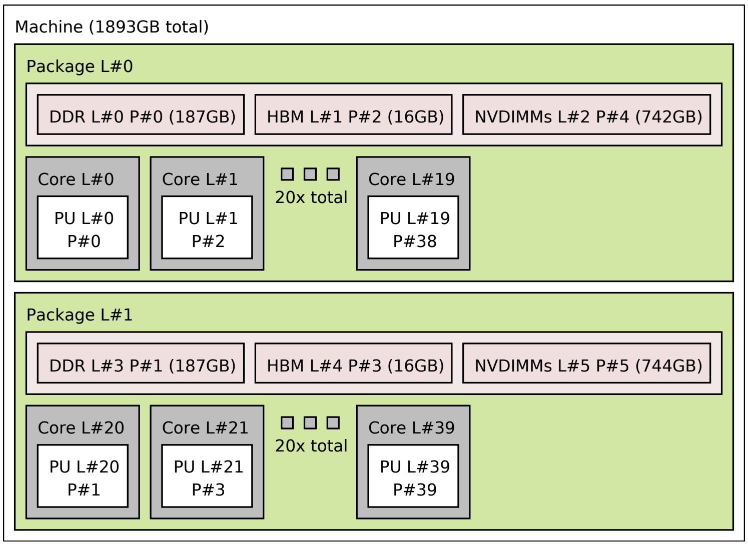


Figure 3.5: hwloc's output of a 3 memory kind machine. The CPU exposes DDR memory as NUMA nodes, NVDIMM memory and HBM memory are shown as extra NUMA nodes (cache hierarchy hidden).

attaching multiple NUMA nodes near the same group of cores. I.e. CPUs can have more than one local NUMA node, and each corresponding to different memory kinds. Considering this, applications may encounter more than one local NUMA node, making necessary to take into account more parameters to allocate memory buffers in the right place. [1]

3.1.1 Identifying Memories

Identifying memory kinds consist of understanding which NUMA nodes are of which kind. E.g., In the fictitious 3MK platform in Figure 3.5, how an application can know that the first NUMA node is DRAM or that the second one is HBM or the third one is NVDIMM? An unaware application may be confused and think all of them are DRAM instead.

By default, most memory allocations should go to the DRAM because its capacity is high enough and its performance is reasonable enough. On contrary, the low HBM capacity should be used only for performance-critical allocations in Figure 2.23. NVDIMMs should only be used for large non-performance-critical buffers. Hence, DRAM should likely be exposed to the application first, before NVDIMM and HBM if they exist in the platform. However, the ranking of NUMA nodes exposed to applications depends on the operating system and the hardware ACPI tables. In consequence, applications and runtimes developed for KNL, for instance, memkind [16] included some KNL-specific detection code to hardwire the ranking of DRAM and HBM NUMA nodes. High-productivity on modern platforms requires a portable solution: a way to find out which nodes are DRAM, HBM or NVDIMM without knowing in advance how they may be exposed by the hardware and operating system.

3.1.2 Characterising Memories

Current hardware specifications (ACPI tables) and operating systems do not explicitly expose information about the memory kind behind each NUMA node. One reason is that vendors do not want developers to assume that NVDIMM is slower than DRAM because it is not always true (NVDIMM-F has DRAM performance) [2]. **Hence, we propose to base identification on the performance characterisation of memory nodes**. For instance, instead of requesting an allocation on HBM, let the application allocate on the memory node with the best bandwidth. It is more portable because applications do not assume that a given platform has HBM or not. It only requests the best available one. Thus, there is a need to characterise the memory nodes through metrics that are relevant to application needs, for instance, low latency, high bandwidth or high capacity.

[1] By default, hwloc exposes DRAM first because it is usually the default allocation node.

[2] https://lkml.org/lkml/2019/3/25/1020

hwloc can perform the identification by looking at which drivers manage each device, however, the identification is only used for human debugging, for instance verifying that the allocation went to HBM on a specific platform where the application requested a high-bandwidth allocation.

3.2 Memory Attributes

In our work, we assign a set of attributes to each memory device, given the heterogeneity of memory systems. These attributes help us to classify them in terms of intuitive metrics to help the selection of the right memory device given a use case. It gave us the ability to create an ordering of memories for a given attribute. The best, second best, ..., and if the best memory devices are available, an application may choose it to allocate their memory buffers.

3.2.1 Bandwidth

Computer architectures such as KNL include two types of memory, high-capacity memory DRAM and high bandwidth memory HBM. Our Kona01 platform is one example of this. There is a large bandwidth differential between these two (four to five times), which makes bandwidth-intensive applications extremely sensitive to the choice of memory and hardware mode [95] (e.g., Cache and Flat modes). These applications should be able to allocate data buffers in the most appropriate memory.

HBM play a key role in future systems. Currently, it is a crucial component on systems such as Summit and Sierra from the U.S. Department of Energy [104] (coming back in next generation Intel Xeon "Saphire Rapids") and Fugaku from Riken [51].

We propose to characterise memory targets with a bandwidth value or even one for reading or one for writing if necessary. These values allow to have an ordering when bandwidth is the priority criterion for an allocation.

On the Leonide platform, we have either DRAM and NVDIMMs, and the resulting ranking based on the bandwidth attribute may return the following ordering:

$$DRAM_{BW} > NVDIMM_{BW} \tag{3.1}$$

The ordering puts DRAM as the memory with the highest bandwidth.

On the Kona01 platform, the resulting ranking based on the bandwidth attribute may return the following ordering:

$$HBM_{BW} > DRAM_{BW} \tag{3.2}$$

Here, the result evidence that HBM is the memory with the best bandwidth capabilities.

On a 3MK HMS system having DRAM, HBM and NVDIMMs, the resulting ranking based on the bandwidth attribute may return the following ordering:

$$HBM_{BW} > DRAM_{BW} > NVDIMM_{BW} \tag{3.3}$$

Here, HBM is ranked before DRAM and DRAM before NVDIMMs.

However, as mentioned before, this attribute could give us more than one order due to the fact that in some cases we could incur in having very different values between reading access and write access. This is theoretically possible but unlikely.

3.2.2 Latency

Although seemingly related, bandwidth and latency may not be correlated in practice. On platforms with fast and slow memory such as our three platforms, one may expect the fast memory to always provide better latency and bandwidth than the slow memory. This may not be the case. For instance, in our 2MK HMS system, Kona01 HBM latency is higher than DRAM's when the platform is loaded. This is very important for HPC nowadays [65]. Moreover, application requirements may depend specifically on bandwidth (like streaming kernels) or latency (like pointer chasing-like applications). Thus, providing independent attributes for bandwidth and latency allows to better describe application needs.

Hence, in the same manner as bandwidth, we have characterised memories by latency taking into account the probable separation of reading and writing latencies. This attribute would lead, when the latency is the priority criterion and as result, we could have the following orderings for our platforms [3].

On the Leonide system, the resulting ranking based on the latency attribute may return the following ordering:

$$DRAM_{Lat} > NVDIMM_{Lat} \tag{3.4}$$

The resulting ordering puts DRAM as the memory with the lowest latency.

On the Kona01 KNL system, the resulting ranking based on the latency attribute may return the following ordering:

$$DRAM_{Lat} \simeq HBM_{Lat} \tag{3.5}$$

[3] This order compares the priority of memories for latency-sensitive allocations, and not the latency itself (the weaker latency is the more priority that memory has, the further left it appear in our equation)

Here, applications on KNL will not know where to allocate on DRAM or HBM since the priority is similar. In this case, it should be necessary to look at other criteria, such as the capacity to finalise its choice.

On a 3MK HMS system having DRAM, HBM and NVDIMMs, the resulting ranking based on the latency attribute return the following ordering:

$$DRAM_{Lat} \simeq HBM_{Lat} > NVDIMM_{Lat} \tag{3.6}$$

Inside our fictitious platform, we have considered that the relation between DRAM and HBM would be pretty similar to Kona01 behaviour (but it could not be the case in future systems). That is, that the resulting ranking could need the support of another attribute as capacity.

3.2.3 Capacity

The capacity of a memory device is one of the best-known attributes that allows us to have an important criterion when deciding where to allocate the buffers. This decision is particularly relevant when the heterogeneous memory system contains memory kinds with limited capacity. An example of this is shown in Kona01, which has an HBM memory of just 16GB compared to 96GB of DRAM. On the other hand, we can also find capacity-intensive applications that obviously will not be able to use the HBM, but that could even have limitations with the capacity of the DRAMs. An example of this could happen in our HPC 2MK HMS system Leonide, where the NVDIMMs are very useful.

On Leonide system, the resulting ranking based on the capacity attribute may return the following ordering:

$$NVDIMM_{Cap} > DRAM_{Cap} \tag{3.7}$$

The resulting ordering puts NVDIMM as the memory with the highest capacity.

On Kona01 system, the resulting ranking based on the capacity attribute may return the following ordering:

$$DRAM_{Cap} > HBM_{Cap} \tag{3.8}$$

On a 3MK HMS system, the resulting ranking based on the capacity attribute returns the following ordering:

$$NVDIMM_{Cap} > DRAM_{Cap} > HBM_{Cap} \tag{3.9}$$

The importance of exposing the heterogeneous memory system using different attributes allows us to build or identify the memory characteristics of a given system. In Figures 3.6, 3.7 and 3.8 we can observe respectively the memory characteristics of Leonide, Kona01 and 3MK respectively.

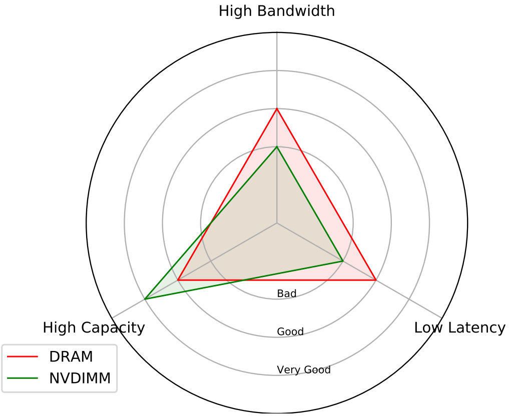


Figure 3.6: Memory attributes characteristics of Leonide.

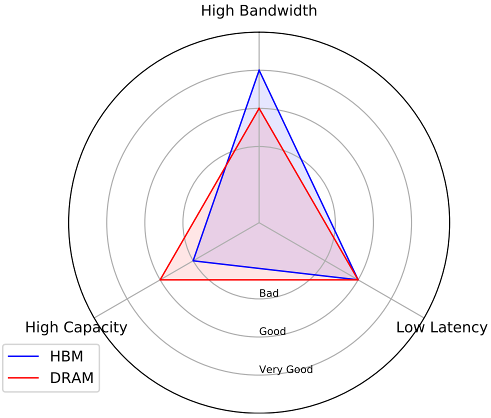


Figure 3.7: Memory attributes characteristics of Kona01.

3.2.4 Locality

We also envision a locality attribute that would describe whether a memory device is attached specifically to a subset of cores or shared by many of them. In Figure 2.22, HBM has strong quad-core locality because it is only attached to one SubNUMA Cluster. DRAM and NVDIMM are local to twice as many cores (entire CPU package), and the network-attached memory is shared by the entire machine.

The locality attribute can be useful when it comes to share data between cores: two tasks sharing data might perform better if data is stored on local

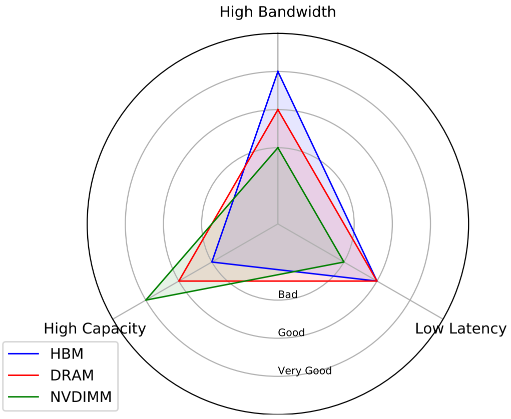


Figure 3.8: Memory attributes characteristics of our 3MK HMS node.

memory device for both and if the interconnection of the memory is in dispute. Hence, this could be a filter to avoid non-local memory devices for the cores involved. A use case of this comes when running an MPI application on a machine with a topology similar to Figure 2.22 in where it can create MPI sub-communicators based on the locality attribute. When the distance is 1, MPI tasks running on the same SubNUMA Cluster are assigned to the same sub-communicator and to the same HBM local memory. When the distance is 2, MPI tasks running on the same package are assigned to the same sub-communicator and to the same DRAM memory. Note that similar groupings can be created today using hwloc's compute devices (e.g., Package). In contrast, our locality attribute targets the memory devices in the machine. This difference may be significant for two reasons: (1) Architectures may not have a 1:1 correspondence between compute packages and memory, and (2) our approach would provide a memory handle to the local memory based on the given distance–without having to specify what NUMA domain that may be, if any.

3.2.5 Other Attributes

We have limited mainly this study to the attributes of capacity, latency and bandwidth. However, the characterisation of a memory kind is not limited to these attributes. As shown in Table 3.1 many other attributes can be supported. As mentioned, separate bandwidth or latency for reads an writes, allows to have **Read Bandwidth, Write Bandwidth, Read Latency and Write Latency** as additional attributes. Those are being considered for addition when the relevant hardware information is available, for instance **power consumption**, the **endurance** or the **persistence** of NVDIMMs.

Read Bandwidth, Write Bandwidth, Read Latency and Write Latency attributes are important especially for NVDIMMs, where there are very big differences comparing reads and writes. E.g., On Leonide, Figure 3.9 shows the output of the Locality Aware Roofline Model (LARM) [20] which reports bandwidth differences between Load, Store and non-temporal stores (StoreNT).

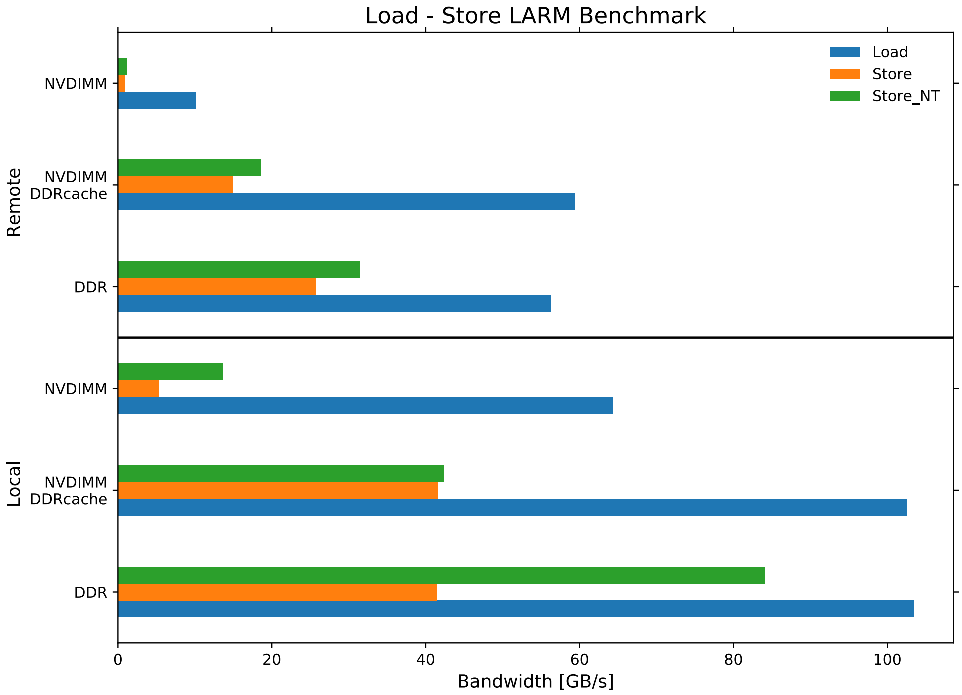


Figure 3.9: LARM Write/Read Bandwidth report on Leonide.

The power attribute, nowadays is a very important criterion to be considered for data-placement algorithms [76]. Endurance attribute gives an idea of economise the use of NVDIMMs due to long-times and to a certain extent, work mainly on DRAM and then store the final information on NVDIMM. Persistence attribute that basically makes reference to critical information that should not be lost during a power cut.

Table 3.1: Status of memory attributes.

Attributes	Native Discovery
Capacity, Locality	Always supported
Bandwidth, Latency	On **most** platforms
R/W Bandwidth, Latency	On **some** platforms
Persistence, Endurance, Power	Under investigation
Custom Metrics	N/A

3.3 Implementation in hwloc

As was mentioned in Section 2.5.2 buffers by default are allocated on DRAM memory regardless of the platform, because the OS and hwloc expose them first. At the beginning as shown in the left part of Figure 3.10, CPUs were not able to be part of 2 NUMA nodes.

Nowadays, heterogeneous memory systems allow to have more than one memory kind local to a CPU. It breaks the old concept of NUMA node elements, having changed from containing CPUs and the memory to having CPUs and several local memories. For that reason it is more proper to talk about a relation of an initiator and a target such as the ACPI specifications does. In where, initiators are either a set of logical processors (CPU-sets) or specific objects, and targets are hwloc objects of type NUMA node.

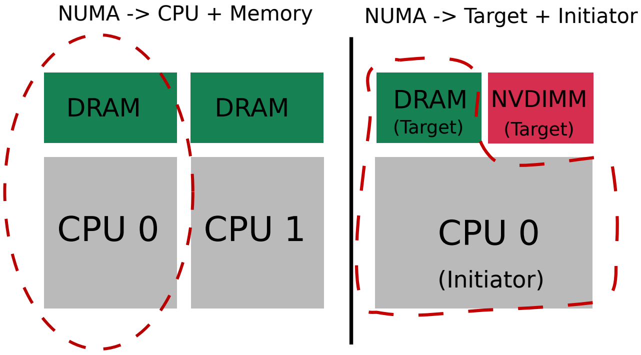


Figure 3.10: NUMA nodes redefinition.

Our API showed in Figure 3.11 implements the aforementioned ideas within hwloc. It extends the already available support for the memory systems allowing hwloc to expose memory characteristics and orderings.

An application using the interface will normally select the targets that are local to the core(s) where it runs (NUMA affinity), and after compares their values for some attributes. One attribute (e.g. bandwidth, latency or capacity) will correspond to the priority criteria when deciding where to allocate. This API works also in homogeneous NUMA platforms since bandwidth and latency indicate whether NUMA nodes are close or far away from cores.

One manner to use the API consists in starting from the core(s), where an application is bound and finding the best target for allocating memory nearby. This may be obtained by passing the related core(s) *cpuset* as an initiator to `get_best_target(...)` with the relevant **memory attribute** such as shown in Figure 3.12. E.g if the applications is bandwidth-bound, use the Bandwidth attribute.

The API also supports cases where an allocations fails, e.g. due to a lack of capacity, or when more complex decisions are needed, or even if non-local memory may be needed.

Get the array of memory targets that are local to a given initiator:

`hwloc_get_local_numanode_objs(topology, initiator, &nr, &targets)`

Get the best memory target (and its value) for the given initiator and attribute:

`hwloc_memattr_get_best_target(topology, attribute, initiator, &best_target, &target_value)`

Get the value of an attribute for the given memory target and initiator:

`hwloc_memattr_get_value(topology, attribute, target, initiator, &value)`

Add a specific memory attribute:

`hwloc_memattr_register(topology, attribute, name, &value)`

Figure 3.11: Summary of main hwloc API functions for manipulating memory attributes. Initiators are either sets of logical processors (CPU-set) or specific objects. Targets are hwloc *objects* of type NUMA node.

```
/* Initialise Topology */
hwloc_topology_init(&topology);
hwloc_topology_load(topology);

[...]

/* Allocating function */
void * alloc_on_best_target(topology, initiator, attribute, size)
{
  hwloc_memattr_get_best_target(topology, attribute, initiator,
                               &best_target, NULL);
  return hwloc_alloc_membind(topology, size, best_target->nodeset,
                             BIND, BYNODESET);
}

[...]

/* Allocating 1MB on best bandwidth memory near a given core */
void * buffer = alloc_on_best_target(topology, core->cpuset,
                                    HWLOC_MEMATTR_ID_BANDWIDTH,
                                    1024*1024);
```

Figure 3.12: Example of hwloc API use to allocate on the best target for an existing attribute.

Figure 3.13 represents a more complex problem where somebody wants a custom metric for the STREAM-Triad kernel (2 reads for 1 write). We first obtain a list of local NUMA nodes by passing the initiator (cpuset) to `get_local_numanode_objs(...)`. Then we create the custom attribute with `register(...)` and manually compute its value for different targets by combining Read an Write bandwidths obtained the `get_value(...)`. Finally, the new attribute may be used to allocate with the previously defined allocation routine in Figure 3.12.

```
/* Initialise Topology */
hwloc_topology_init(&topology);
hwloc_topology_load(topology);

/* Register Custom Attribute */
hwloc_memattr_register(topology, attribute, name, &customMetric, flags);

/* Get array of targets given an initiator */
hwloc_memattr_get_local_numanode_objs(topology, initiator,
                                     &nr, &targets);

/* Initialise Custom Metric Values */
foreach(target in targets) {
  /* Get Read/Write Bandwidth values */
  hwloc_memattr_get_value(topology, ReadBandwidth, target, initiator,
                         &rbw_value);
  hwloc_memattr_get_value(topology, WriteBandwidth, target, initiator,
                         &wbw_value);

  /* Store the metric value for this node by combining R/W bandwidths */
  custom_metric_value = rbw_value||rbw_value||wbw_value;
  hwloc_memattr_set_value(topology, customMetric, target, initiator,
                          flags, custom_metric_value);
}

/* Allocating on best node for the custom metric */
buffer = alloc_on_best_target(topology, initiator, customMetric, size);
```

Figure 3.13: Defining a custom metric for STREAM-Triad kernel (2xReadBW+1xWriteBW) and allocating in the best target for that metric.

Figure 3.14 shows later in this chapter an example of output of this API. hwloc was already able to expose some default attributes coming from the operating system (such as capacity and locality). However, not all platforms present reliable values about latency and bandwidth and for this reason next section shows the available options to obtain this values.

3.4 Attributes Values

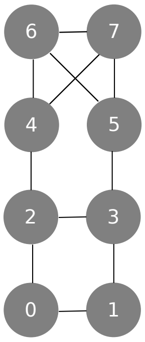
Discovering attribute values is another task that consist of characterising the memory targets. There are two sources that can be used to obtain this information. The first is use hardware-provided information, in our case via hwloc by accessing to Heterogeneous Memory Attributes Table (HMAT) included in the 6.2 revision of the ACPI specification. This table is expected to be generalised in the next platforms with reliable information. Until that happens, the values of the attributes could be obtained by benchmarking or measuring each of them. There are many benchmarks that can be used, e.g., the STREAM-Triad could give us informations about bandwidth under different access patterns.

3.4.1 ACPI SLIT

The ACPI System Locality Information Table (SLIT) provides information about relative differences in access latencies between the CPU from one NUMA node to the memory of another. Unfortunately this table was not heavily used by software hence hardware vendors often provided very simple or even invalid latency information. In the end, the use of this table was mostly to differentiate what is local and what is not. E.g., the topology shown in Table 3.2a represents a 8 AMD Opteron machine, where latency depends of the distance [64]. However, in Table 3.2b shows that the ACPI SLIT only says 10 for local and 20 for remote, while it should report several different values depending on the physical node distances.

Table 3.2: 8 AMD Opteron with the motherboard TyanS4881+M4881.

(a) Topology.



(b) ACPI SLIT.

node	0	1	2	3	4	5	6	7
0	10	20	20	20	20	20	20	20
1	20	10	20	20	20	20	20	20
2	20	20	10	20	20	20	20	20
3	20	20	20	10	20	20	20	20
4	20	20	20	20	10	20	20	20
5	20	20	20	20	20	10	20	20
6	20	20	20	20	20	20	10	20
7	20	20	20	20	20	20	20	10

In Table 3.3 we can observe that SLIT does not expose the memory heterogeneous system properly on Kona03 (see Appendix A.1.2). Indeed Intel used special values so that the OS does not allocate by default on HBM memory. The corresponding values are 10 for CPU to local DDR (or none CPU to local HBM), 21 for CPU to remote DDR, 31 for CPU to local HBM and 41 for

CPU to remote HBM (or non CPU to remote HBM). In practice, the latency of local HBM (31 in SLIT) should be smaller than the latency of remote DDR (21 in SLIT). Hence applications and runtimes such as memkind were explicitly modified to detect this strange table to identify the KNL configuration.

Table 3.3: ACPI SLIT of Kona03, nodes 0-3 are DRAM and nodes 4-7 are MCDRAM.

node	0	1	2	3	4	5	6	7
0	10	21	21	21	31	41	41	41
1	21	10	21	21	41	31	41	41
2	21	21	10	21	41	41	31	41
3	21	21	21	10	41	41	41	31
4	31	41	41	41	10	41	41	41
5	41	31	41	41	41	10	41	41
6	41	41	31	41	41	41	10	41
7	41	41	41	31	41	41	41	10

With nowadays platforms where a single CPU initiator may have different local memory targets, the SLIT ACPI table cannot represent the system topology correctly because the notion of NUMA nodes composed of CPUs and memory is obsolete. Moreover, bandwidth is often considered as very important, not only latency, for placement of data buffers. Hence, vendors worked on updating the ACPI specification.

3.4.2 ACPI HMAT

The Heterogeneous Memory Attributes Table (HMAT) has been introduced in the revision 6.2 of ACPI specification to modernise the way the hardware provides the software with a description of the memory subsystem. This include support for both latency and bandwidth, as well as initiators and targets instead of generic NUMA nodes. As Figure 3.15 HMAT is formed with the folowing structures [107]:

1. **Memory Proximity Domain Attributes Structure(s)**: Allows to list targets with the corresponding memory address range.

2. **System Locality, Latency and Bandwidth Information Structure(s)**: Contains the actual performance attribute that we use.

3. **Memory Side Cache Information Structure(s)**: Used for exposing the case where DDR is a hardware cache in front of NVDIMM (detailed in Section 6.2.1).

This table should be available on future platforms to describe complex memory hierarchies. Inside HMAT vendors can expose the theoretical latency and bandwidth between initiators (CPU-sets) and all the memory targets(NUMA nodes). We contributed to the exposure of this tables in the sysfs virtual file system starting in Linux 5.2 [4]. We then implemented in this thesis the gathering of these values in hwloc. E.g., Figure 3.14 shows the output of this options on Leonide in where Intel has fulfilled the HMAT.

```
$ lstopo --memattrs
Memory attribute #0 name ‘Capacity’
NUMANode L#0 = 99786076160
NUMANode L#1 = 101468516352
NUMANode L#2 = 796716433408
NUMANode L#3 = 99883061248
NUMANode L#4 = 101428244480
NUMANode L#5 = 798863917056
Memory attribute #2 name ‘Bandwidth’
NUMANode L#0 = 131072 from Group0 L#0
NUMANode L#1 = 131072 from Group0 L#1
NUMANode L#2 =  78644 from Package L#0
NUMANode L#3 = 131072 from Group0 L#2
NUMANode L#4 = 131072 from Group0 L#3
NUMANode L#5 =  78644 from Package L#1
Memory attribute #3 name ‘Latency’
NUMANode L#0 = 26 from Group0 L#0
NUMANode L#1 = 26 from Group0 L#1
NUMANode L#2 = 77 from Package L#0
NUMANode L#3 = 26 from Group0 L#2
NUMANode L#4 = 26 from Group0 L#3
NUMANode L#5 = 77 from Package L#1
```

Figure 3.14: Extracts from hwloc’s lstopo reporting memory attributes on the HMAT at the Xeon platform depicted by Figure 3.16.

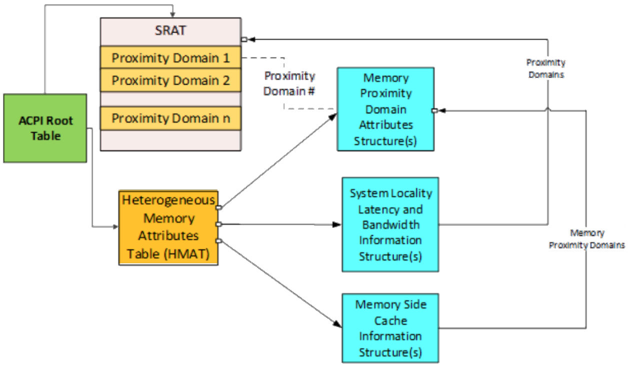

SRAT
Proximity Domain 1
Proximity Domain 2
Proximity Domain n
ACPI Root Table
Proximity Domain #
Memory Proximity Domain Attributes Structure(s)
Proximity Domains
Heterogeneous Memory Attributes Table (HMAT)
System Locality Latency and Bandwidth Information Structure(s)
Memory Proximity Domains
Memory Side Cache Information Structure(s)

Figure 3.15: HMAT representation.

[4] Unfortunately this is currently limited to the performance of local accesses

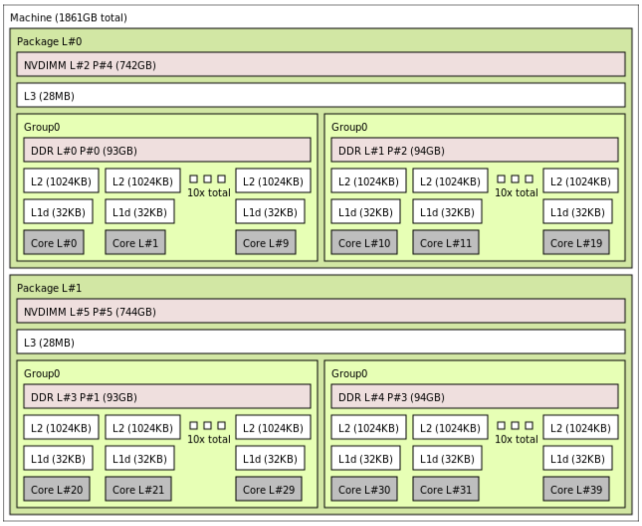


Figure 3.16: Output of hwloc's lstopo tool on a dual Xeon 6230 with 384GB of DRAM (96GB per *SubNUMA Cluster* of 10 cores) and 1.5TB of NVDIMMs (768GB per CPU). NVDIMMs are configured in *1-Level-Memory* and exposed to applications as additional NUMA nodes.

3.4.3 Benchmarking

Even though HMAT seems to be a good idea to describe memory attribute values, few machines are implementing it yet[5] and there is still a risk of vendors not implementing them correctly[6]. Until HMAT is effectively properly implemented in all platforms, hwloc may use experimentally measured attribute values. Moreover, ACPI tables do not provide informations about every attribute. E.g., separate values for reads and writes are optional.

Discovering memory attribute values (whatever it is) through benchmarking gives developers the ability to not have to wait for vendors to provide HMAT tables. However, doing this usually takes a great deal of time. Despite this disadvantage, this act should really only be performed once.

3.4.3.1 Latency Benchmarking Experiments

Memory latency checker (MLC) is designed to evaluate latencies on Intel platforms [87], including modern servers with NVDIMMs in System-RAM mode since version 3.9. [105] Table 3.4 [105] shows latencies measured on Leonide, which may be used for our latency attribute values. It matches the expected memory characteristics exposed in Figure 3.6. Google Multichase gives the

[5] Things are expected to improve in Ice Lake platforms.

[6] We hope vendors will implement HMAT better than SLIT because the impact of buggy HMAT on performance may be higher, for instance if the operating system does not know where to allocate by default.

Table 3.4: MLC latency evaluation for local and remote targets relative to the initiator 0 in Leonide platform.

	Target			
Initiator=0	0 Local DDR	1 Remote DDR	2 Local NVDIMM	3 Remote NVDIMM
Latency in ns	84.2	145.5	149.7	349.6

Table 3.5: MLC latency evaluation for local and remote targets relative to the initiator 0 in Kona01 platform.

	Target	
Initiator=0	0 Local DDR	1 Local HBM
Latency in ns	117.3	132.6

same behaviour between NVDIMMs and DDR, such as Figure shows 3.17.

In Kona01, Table 3.5 shows that DDR latency is slightly better than HBM. Google Multichase could give a different behaviour depending on the pointer-chase prefetcher mechanism [96] evaluated, in Figure 3.18. In Kona03, we can observe latency behaviour described in Table 3.6, having remotes nodes with HBM memory.

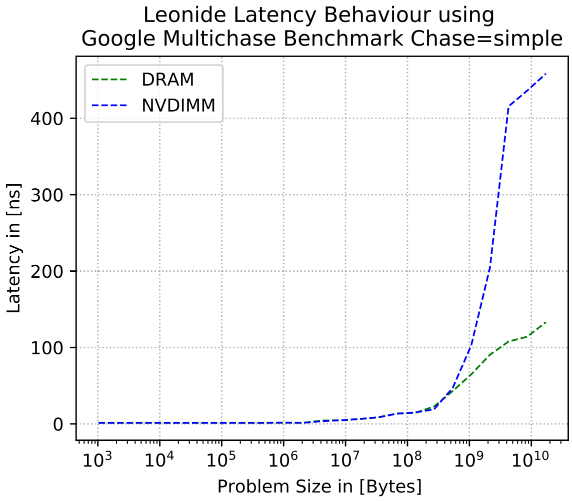


Figure 3.17: Google Multichase latency evaluation of Leonide with a simple chase using one thread per core (20 en total).

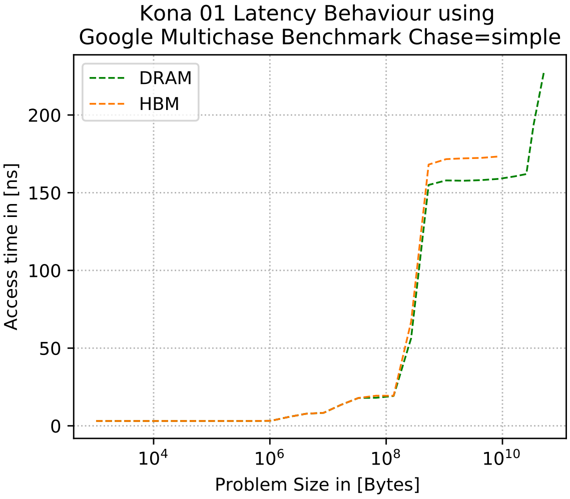


Figure 3.18: Google Multichase latency evaluation of Kona01 with a simple chase using one thread per core (64 en total).

Table 3.6: MLC latency evaluation for local and remote targets relative to the initiator 0 in Kona03 platform.

	Target							
Initiator=0	0 Local DDR	1 Remote DDR	2 Remote DDR	3 Remote DDR	4 Remote HBM	5 Remote HBM	6 Remote HBM	7 Remote HBM
Package	0	1	2	3	0	1	2	3
Latency in ns	114.0	122.1	115.7	123.9	129.1	136.4	133.7	140.9

3.4.3.2 Bandwidth Benchmarking Experiences

There are many benchmarks that can be used to measure bandwidth. For instance, STREAM [61] is a simple benchmark, designed to measure memory bandwidth with four different simple vector kernels. Triad is the most complex of them and is considered highly relevant to HPC. We have measured the benchmark of our systems with the STREAM-Triad with the following conditions: not using hyperthreading, setting the number of OpenMP threads to the number of cores related to the first CPU. Kona01 in Figure 3.19 shows that using the 64 threads (one per core) with HBM gives around 96GB/s in comparison to DRAM with around 60GB/s. Leonide in Figure 3.20 using 20 threads (one per core) gives around 75GB/s when using DDR and as expected NVDIMMs gives only 9GB/s.

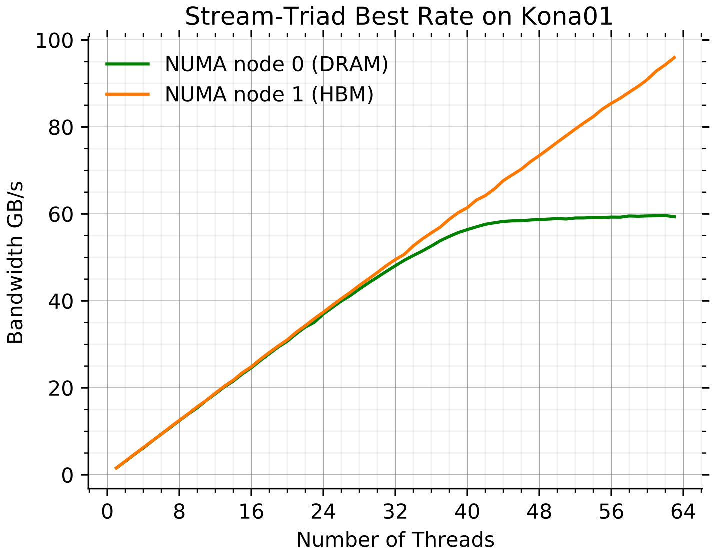


Figure 3.19: STREAM-Triad bandwidth using 2 different kinds of memories in Kona01 with different number of local threads.

These values allow to have a clearer view about the ordering of memories that we pretended to have before when considering bandwitdth as an attribute.

Different benchmarks can be used to measure a certain attribute. This normally generates a variety of results, since in each benchmark measures attributes in a different manner. However, the rankings remain the same. Moreover absolute values are likely meaningless for real applications. In fact, they just need to know which memory is better than another for a given attribute.

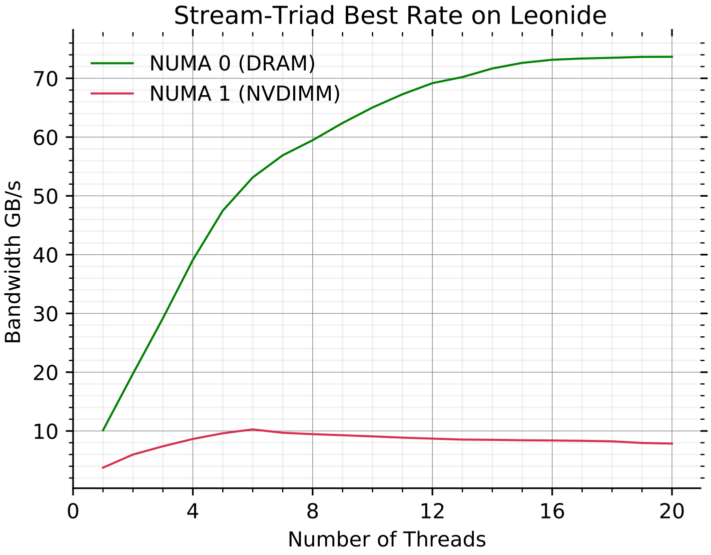


Figure 3.20: STREAM-Triad bandwidth using 2 different kinds of memories in Leonide with different number of local threads.

3.5 Summary

We have presented an interfate to help managing the complexity of the memory system of emerging and future architecures. Due to the different types of memory and their different characteristics, it is a great challenge to use them effectively and efficiently. Ideally, one would like to leverage the low-latency of DRAM, the high-capacity of NVDIMM, and the high-bandwidth of HBM as a single memory system that applications can utilise. This chapter takes a step in that direction by providing building blocks to characterise the heterogeneous memory present on a single machine.

This approach focuses on specifying a number of memory attributes and an API to query and classify the memory devices. These attributes represent high-level characteristics that are relatively easy to reason about bandwidth, latency and capacity. With this interface we expect to simplify the fact of building higher level abstractions to enable to developpers productivity and performance.

Chapter 4

Preparing HPC Applications to Complex Heterogeneous Memory Systems

An unfortunate consequence of the evolution and increasing heterogeneity of memory systems is the need to adapt HPC applications so that they can properly exploit the memory system. A first step has been achieved on the previous Chapter 3, where the memory system has been characterised and exposed for the good use of developers.

However, this does not answer what considerations the developer should have when working with an application with memory-intensive loads. How can he or she know if an application or even its internal buffers have a certain affinity towards some memory attribute (which in turn could refer to a specific type of memory), in other words, to know if the attribute is critical for performance.

This chapter seeks to present our approach to determine a strategy that allows to identify when an application has a certain preference to use a specific type of memory. First by presenting an heterogeneous memory allocator that abstracts in a higher-level the API provided in Chapter 3. Then, analysing some strategies to find a better criteria about where to allocate. Finally, presenting some use case of the presented strategies.

The idea of preparing HPC applications to work in heterogeneous memory systems had a first touch with KNL through AutoHBW [16], where it proposed to allocate buffers in HBM or DRAM depending on the size without having to modify the application code. However, this solution requires to identify the capacity of sensitive buffers for a specific run. Memkind [16] proposes an API specifically designed for KNL that allows to allocate explicitly in fast or slow memory (HBM or DRAM).

Some other strategies have been proposed, taking into account the access pattern of applications. Servat [92] and MOCA [68] use a post-mortem analysis of memory accesses based, e.g., on hardware counters. FlexMalloc [76] proposes to replace dynamic allocations at runtime. And SICM [48] proposes low-level allocation API and high level data management interface by previously performing an Architecture Profiling step to guess the memory organisation.

All these approaches require previous knowledge of which NUMA nodes are fast before adapting the memory allocations. We have prepared this step in Chapter 3 to expose performance characteristics.

We consider applications as a set of memory buffers that must be allocated somewhere. Each buffer may lead to different performance when allocated in different kinds of memory. Therefore, they have an affinity for one or more kinds of memory. In the same fashion they may have an affinity for some location because of which CPU accesses them. This requires to analyse the application behaviour to determine the affinity of memory buffers.

4.1 Heterogeneous Memory Allocator

hwloc through the interface presented in Chapter 3 exposes metric of different memory targets either from hardware or from benchmarking. They may be queried by applications either as explicit values or as a ranking of best targets for some criteria. However, applications using this new low-level hwloc API require a significant rework for end-users even if it consists in just replacing `malloc` with an allocation call on the preferred memory target (beside of manipulating hwloc targets and initiators, one has to first initialise the hwloc topology, etc.). Therefore, we provide a high-level memory allocator for simple use-cases in applications.

We expect the hwloc API to be used in more advanced runtime systems, when using more complex allocations criteria, e.g., handling local and remote memory, migrating memory or even handling specialised attributes that characterise specific access patterns.

The simplified memory allocator is summarised with a single function `mem_alloc(..., attribute)` which basically allows applications to allocate on the best local memory target given a memory attribute such as bandwidth, latency or capacity as listed in Table 3.1. According to the ranking for a specific attribute if the best target is full, the allocator can easily use the second memory target, third memory target, ... on the list. If the attribute is not available on the platform, the allocator may also fallback to other similar attributes, for instance latency instead of read latency.

This first step is an easy and productive modification of applications allocation calls towards more performance. The literature proposes similar solutions, for example, memkind [16]. However, the main difference with our approach is that, for us, the memory attribute specifies what is important for the application (e.g. Bandwidth) without hardwiring to a specific kind of memory (e.g. HBM) as memkind does. Our approach is more portable since it may for instance return the best target found depending of the platform. E.g. a platform as Leonide (not containing HBM) returns DRAM if we consider bandwidth as the memory attribute to be evaluated. Table 4.1 shows the behaviour of our allocator evaluated with different memory attributes and our three different heterogeneous memory platforms used.

Table 4.1: Best resulting targets of using `mem_alloc` with different platforms.

	Leonide	Kona 01	3MK
Default	DRAM	DRAM	DRAM
Latency-Bound	DRAM	DRAM	DRAM
Bandwidth-Bound	DRAM	HBM	HBM
Capacity-Bound	NVDIMM	DRAM	NVDIMM

The heterogeneous allocator modifies each memory allocations by specifying the attribute that responds better to memory buffer requirements. Although, it would seem that code modification is necessary, this step could be avoided by the use of interception techniques and recognising allocation calls (in the same manner that auto-hbwmalloc does [92]) and then add sensitivity hints.

4.2 Allocation Criteria

Our strategy implements steps to provide to application developers a high productivity environment for supporting HMS in a portable way. In Figure 4.1 we can observe 2 stages, one above the heterogeneous allocator and another below it.

Below the allocator we can see that it collects the work done in the previous chapter. There is shown that through hwloc which feeds on with the information that the operating system has can provide information about the hardware (as it did before this thesis) and also on the metrics and information on the performance of the memories, whether they come from ACPI HMAT or of performance measurements. I.e., the interface and allocator already presented offer an easy way for applications to request specific kinds of memory.

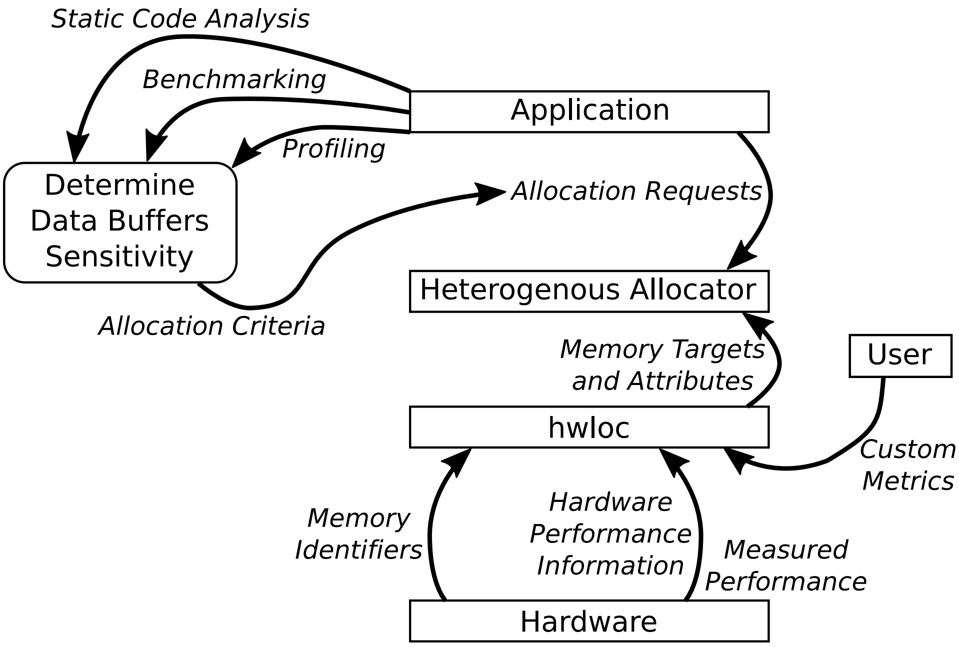


Figure 4.1: General strategy framework.

There are very experienced developers that are able to have a criterion to guess if a buffer is latency-bound or bandwidth-bound, all thanks to years of work optimising cache affinity, prefetching, tiling, etc. That is, there is a level of subjectivity based on the experience of an individual. Given this, there is a need for a more productive framework for non-experts in hardware architectures and code optimisation.

Above the allocator, we look at how to decide what kind of memory an application should request for each buffer. The idea is to analyse the application behaviour to determine the sensitivity of the most important buffers, i.e., buffers whose accesses represent a big part of the execution time.

Figure 4.1 presents three main methods for determining the allocation criteria based on the sensitivities of an application. Benchmarking and Profiling are offline sensitivity detection methods that require a first run for analysis before determining the allocation criteria that can be used in future runs, taking into account that those runs would have a similar behaviour (normally depends on the workloads).

4.2.1 Benchmarking

The simplest strategy for determining sensitivity is to bind the entire process to each kind of memory to then compare the overall performance of each run. This approach only works considering that all buffers of an applications have the same sensibility or if there is a single performance-critical buffer, i.e., that the sensibility will be mostly related to the general performance of an application.

If not, one should rather compare the performance of all possible placements of every buffer in the application, leading to a combinatorial explosion since N buffers imply to M^N possible placement where M is the number of types of memory. For example, Leonide and Kona01 work with a 2MK system and the combinatorial variation correspond to 2^N. In the case of 3MK HMS systems (as Figure 4.2) it corresponds to 3^N. This approach is not very efficient taking into account that very complex applications could have an important number of buffers. For this reason, N might be reduced by identifying buffers that are obviously not performance critical, however, it would add an extra step inside our benchmarking strategy.

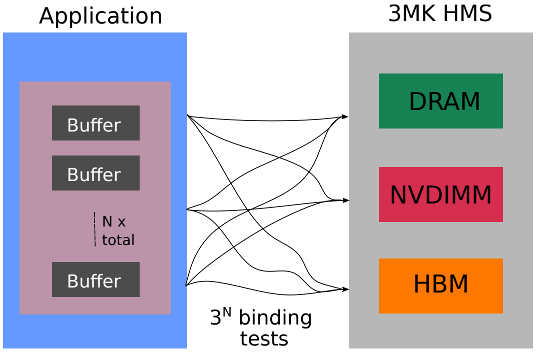


Figure 4.2: Detecting sensitivities by benchmarking an application on 3MK systems.

The benchmarking approach to determine sensitivities is still used extensively in the literature for showing the sensitivity of some benchmarks to certain attributes [77]. Kernels with regular access patterns and streamed accesses as shown in STREAM-Triad benchmark [61, 60] often show greater sensitivity to bandwidth; that is the case shown in Figure 3.19 that tests STREAM-Triad in Kona01 and Figure 3.20 corresponding to Leonide.

Going deeper in the case of STREAM-Triad on Leonide, we can observe in Table 4.2 that the impact of moving buffer A from DRAM to NVDIMM is relevant and more noticeable than buffers B and C. Also, we can notice that things cannot be just black and white (full binding on DRAM or NVDIMM). If we decide to fully bind on the NVDIMM we can see a bandwidth of 8.50 GB/s. But the simple fact of changing that the buffer A uses DRAM implies a bandwidth of 38.32 GB/s, i.e., binding an entire application is not always the best strategy.

Table 4.2: Allocation impact on STREAM-Triad buffers A, B, C between NUMA 0 (DRAM) and NUMA 1 (NVDIMMM) using Leonide HMS.

Buffer target node			Best Rate in GB/s
A	B	C	Triad
0	0	0	74.97
0	0	1	51.88
0	1	0	55.59
0	1	1	38.32
1	0	0	9.92
1	0	1	9.05
1	1	0	9.16
1	1	1	8.50

4.2.2 Profiling

Profiling is a more complex strategy which performs an analysis of the execution using counters and/or instrumentation to identify in detail memory related issues such as bottlenecks, hot spots, etc [89]. Its main benefit in comparison to benchmarking is that it does not require several runs to identify application sensitivities.

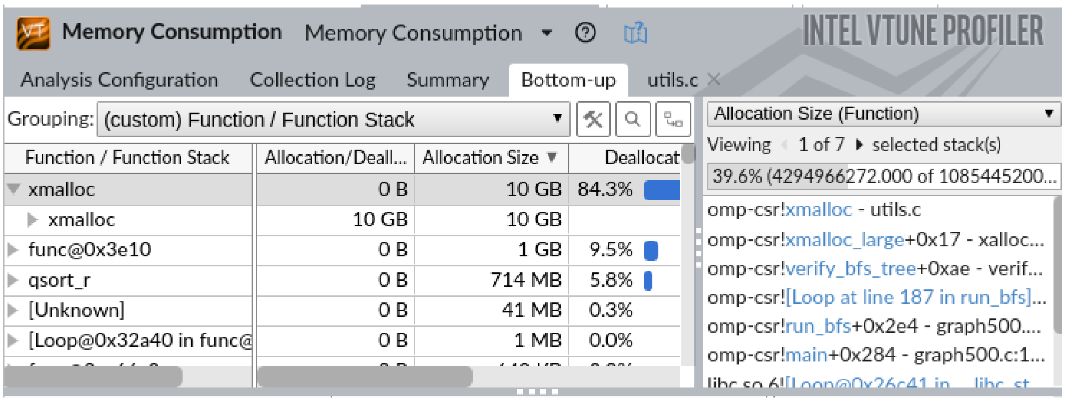
Many tools have been proposed to determine the memory access pattern and to display profiled information graphically. Some of them are even capable to link the execution traces to the corresponding lines of code and memory allocations.

4.2.2.1 Using Intel Vtune Profiler

The Intel VTune Profiler allows analysing a large spectrum of different buffer/application sensitivities such as bandwidth, latency, capacity, R/W bandwidth, persistence and energy [53].

To identify capacity-sensitive buffers inside applications, VTune allows to do a *Memory Consumption Analysis* of the profiled application over the time.

This analysis basically tracks all allocations made by the application. In Figure 4.3 we can observe how in Graph500 application the most consuming buffer of the application corresponding to `xmalloc` with an allocation size of 10 GB. Considering the capacity of DRAM, it could not represent a capacity-sensitive buffer.



Function / Function Stack	Allocation/Deall...	Allocation Size ▼	Deallocat
xmalloc	0 B	10 GB	84.3%
xmalloc	10 GB	10 GB	
func@0x3e10	0 B	1 GB	9.5%
qsort_r	0 B	714 MB	5.8%
[Unknown]	0 B	41 MB	0.3%
[Loop@0x32a40 in func@	0 B	1 MB	0.0%

39.6% (4294966272.000 of 1085445200...

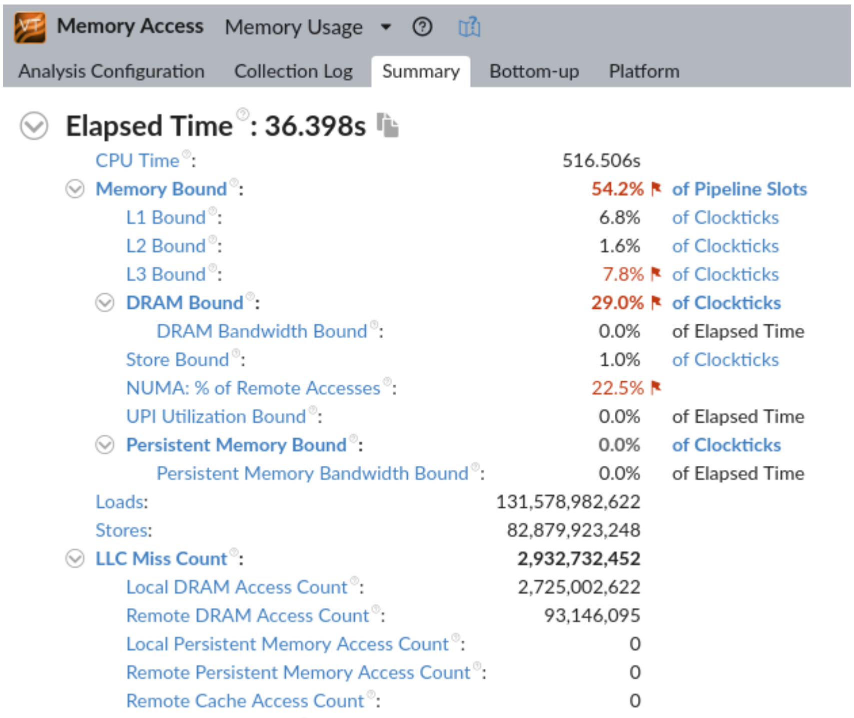
omp-csr!xmalloc - utils.c
omp-csr!xmalloc_large+0x17 - xalloc...
omp-csr!verify_bfs_tree+0xae - verif...
omp-csr![Loop at line 187 in run_bfs]...
omp-csr!run_bfs+0x2e4 - graph500....
omp-csr!main+0x284 - graph500.c:1...

Figure 4.3: Graph500 capacity-sensitivity buffer check with Intel VTune Profiler.

To analyse buffer's bandwidth- and latency-sensitiveness the *Memory Access Analysis* tool should be used to get information about hot memory objects in a program [78] in terms of latency and bandwidth. Figure 4.4 presents *DRAM Bound* and *Persistent Memory Bound* metrics that rely on Intel-specific counters indicating whether many cycles are spent accessing DRAM or NVDIMMs, hence showing CPU stall and latency issues. In relation to bandwidth it presents *DRAM Bandwidth* and *Persistent Memory Bound* metrics that are related to bandwidth issues as a percentage of elapsed time. Furthermore, the analysis is capable to give us information of kind of memory used and the list of buffers ordered by importance. A detailed example of this is detailed in Section 4.3.

4.2.2.2 Other alternatives

The Intel VTune Profiler is not the only tool of Intel designed for profile applications. Intel Advisor XE is a design assistance and analysis tool also for memory use [62]. It automates the Roofline Performance Model proposed at Berkeley, helping to identify if a given loop/function is memory or CPU bound. It permits to do a analysis of data transactions between different memory layers. However, Intel tools are designed to work mainly with Intel technology. 3DyRM is an extension of the roofline model including memory latency information to better represent the behaviour on systems with heterogeneous memory systems [57].



Elapsed Time: 36.398s

CPU Time:	516.506s	
Memory Bound:	54.2% ⚑	of Pipeline Slots
L1 Bound:	6.8%	of Clockticks
L2 Bound:	1.6%	of Clockticks
L3 Bound:	7.8% ⚑	of Clockticks
DRAM Bound:	29.0% ⚑	of Clockticks
DRAM Bandwidth Bound:	0.0%	of Elapsed Time
Store Bound:	1.0%	of Clockticks
NUMA: % of Remote Accesses:	22.5% ⚑	
UPI Utilization Bound:	0.0%	of Elapsed Time
Persistent Memory Bound:	0.0%	of Clockticks
Persistent Memory Bandwidth Bound:	0.0%	of Elapsed Time
Loads:	131,578,982,622	
Stores:	82,879,923,248	
LLC Miss Count:	2,932,732,452	
Local DRAM Access Count:	2,725,002,622	
Remote DRAM Access Count:	93,146,095	
Local Persistent Memory Access Count:	0	
Remote Persistent Memory Access Count:	0	
Remote Cache Access Count:	0	

Figure 4.4: Graph500 latency-, bandwidth-sensitivity check with Intel VTune Profiler.

There are other relevant non-Intel profiling tool alternatives such as: Likwid tools through the command line tool likwid-perfctr [100] allows to evaluate cache and memory bandwidth taking into account the system topology. The Oracle Sampling Collector and Performance Analyser allows to collect performance data by tracing among other things memory allocation and deallocation calls and then display memory metrics of performance of a targeted applications. HPC toolkit permits to analyse the performance of an application with performance memory units [63].

4.2.3 Static Code Analysis

Static code analysis (also listed in Figure 4.1) consists in studying the source code, for example, during compilation. This step can be used for finding bugs or for providing the compiler additional information at runtime about the program, for instance what will to happen in the future with a data buffer.

Compilers have been trying for a long time to reduce memory access latency by inserting software prefetching for specific applications and/or specific platforms [6], [99]. We believe that this type of work should allow compilers to detect latency- or bandwidth-sensitivity of kernels and thus provide sensitivity hints to runtime systems. For instance, streamed/linear accesses to contiguous buffers can be detected and marked as bandwidth sensitive without the need for the user to manually benchmark or profile the application.

This approach has been proposed for different languages to detect frequently used data and memory access patterns, and decide to allocate in HBM or DRAM [45]. Variables are annotated with priority values, and the compiler statically overrides `malloc` calls to allocate high-priority buffers to HBM. In our approach, the idea is that the compiler insert annotations in the code to tell the runtime where to allocate each buffer. This can be done by replacing allocation calls with our heterogeneous allocator. However, providing information to the runtime is generally a more flexible approach, as it allows the runtime to make more informed decisions about the overall execution.

4.3 Use Case

The method presented in Figure 4.1 can be applied to provide memory allocations that respect affinities and needs of computational tasks. This section describes how we have successfully applied it, using the benchmarking and profiling approximations. This use cases have been done in our already mentioned platforms in Chapter 3 corresponding to Leonide and Kona03 both corresponding to a 2MK HMS system.

4.3.1 Benchmarking

We have applied this method using Graph500 application [66] that uses irregular memory accesses [78]. We have used Graph500 version 3.0.0 parallelised with MPI. The performance is measured by an harmonic average of Traversed Edges per Second (TEPSe+8).

In Table 4.3 we can observe Graph500 performance depending on the memory placement of the entire process. On Leonide, DRAM provides results between 1.5 to 3 times higher than NVDIMM. This confirms that this application should specify either latency or bandwidth as a priority in our heterogeneous allocator since Leonide’s DRAM is faster for both metrics. However, on Kona03, DRAM results are too much close to HBM. The Latency of both memories is actually similar while the bandwidth is very different (90GB/s against 350GBs approximately). It shows that the bandwidth criterion is not suitable for this allocation (the gain is too weak to justify consuming the low HBM capacity). These results confirm what was expected: Graph500 application is rather limited by latency because it performs memory accesses with indirections during the graph transversal.

On the other hand, when an application is limited by bandwidth, e.g., STREAM-Triad [61] in Table 4.4, the bandwidth is obviously the criterion that should be passed to our allocator.

Thus, with our allocator and the bandwidth/latency criteria, we are able to allocate memory for two very different architectures. Combined with the capacity criterion, this work allowed to dynamically adapt the allocations according to the needs of applications and to the actual available memory. This is not possible with the existing interfaces because the application would only be able to request HBM explicitly instead of requesting a memory with good latency. HBM allocations are not possible on Leonide (Xeon), while HBM allocations on Kona03 (KNL) would consume HBM without actually needing it for better performance. Our work provides same performance as manual tuning while remaining portable, hence providing superior productivity.

Table 4.3: Graph500 performance in Traversed Edges per Second (TEPSe+8).

Graph Size	DRAM	NVDIMM
2.15 GB	3.423	2.056
4.29 GB	3.459	2.067
8.59 GB	3.481	2.084
17.18 GB	3.343	2.107
34.36 GB	2.990	1.044

(a) Leonide (Xeon): 16 MPI processes on a single processor using its local DRAM or NVDIMM.

Graph Size	HBM	DRAM
2.15 GB	0.418	0.415
4.29 GB	0.402	0.396

(b) Kona03 (KNL): 16 MPI processes on a *SubNUMA Cluster* using its local HBM or DRAM.

Table 4.4: STREAM-Triad throughput in GB/s depending on the optimised criteria. *Best Target* corresponds to the local NUMA node that the allocator found most appropriate for this criteria.

		Total allocated memory for arrays		
Optimised Criteria	Best Target	22.4GiB	89.4GiB	223.5GiB
Capacity	NVDIMM	31.59	10.49	9.46
Latency	DRAM	75.06	75.24	–

(a) Leonide (Xeon) with 20 threads on a single processor using its local DRAM (192GB) or NVDIMM (768GB) in 1-Level-Memory mode.

		Total allocated memory for arrays		
Optimised Criteria	Best Target	1.1GiB	3.4GiB	17.9GiB
Bandwidth	HBM	85.05	89.90	–
Latency	DRAM	29.17	29.17	29.16

(b) Kona03 (KNL) with 16 threads on a *SubNUMA Cluster* using its local HBM (4GB) and DRAM (24GB).

4.3.2 Profiling

Profiling experiments have been done in Leonide platform to further dig into the details of data buffer sensitivity to bandwidth and latency. For the test we have used Graph500 version 2.1.4 and as described before we have used Intel VTune Profiler. We presume that we do not know the latency-sensitivity of Graph500 main buffers and we profile the execution when allocating on DRAM and on NVDIMMs separately. For this test, only cores of a single processor are used together with their local memory. The old version 2.1.4 of Graph500 is used here for OpenMP support, because profiling a single process is easier (however profiling the more recent MPI version is also possible). As seen earlier in Table 4.3a, allocating the entire process on DRAM brings about two times better performance than on NVDIMMs.

First, the summary of the memory access analysis gives information about the overall application sensitivity to latency or bandwidth. Table 4.5 presents the relevant information for our work. For Graph500, VTune shows an indicator flag on the DRAM Bound parameter meaning that the application is latency sensitive, especially when running on NVDIMMs because this memory has a high latency.

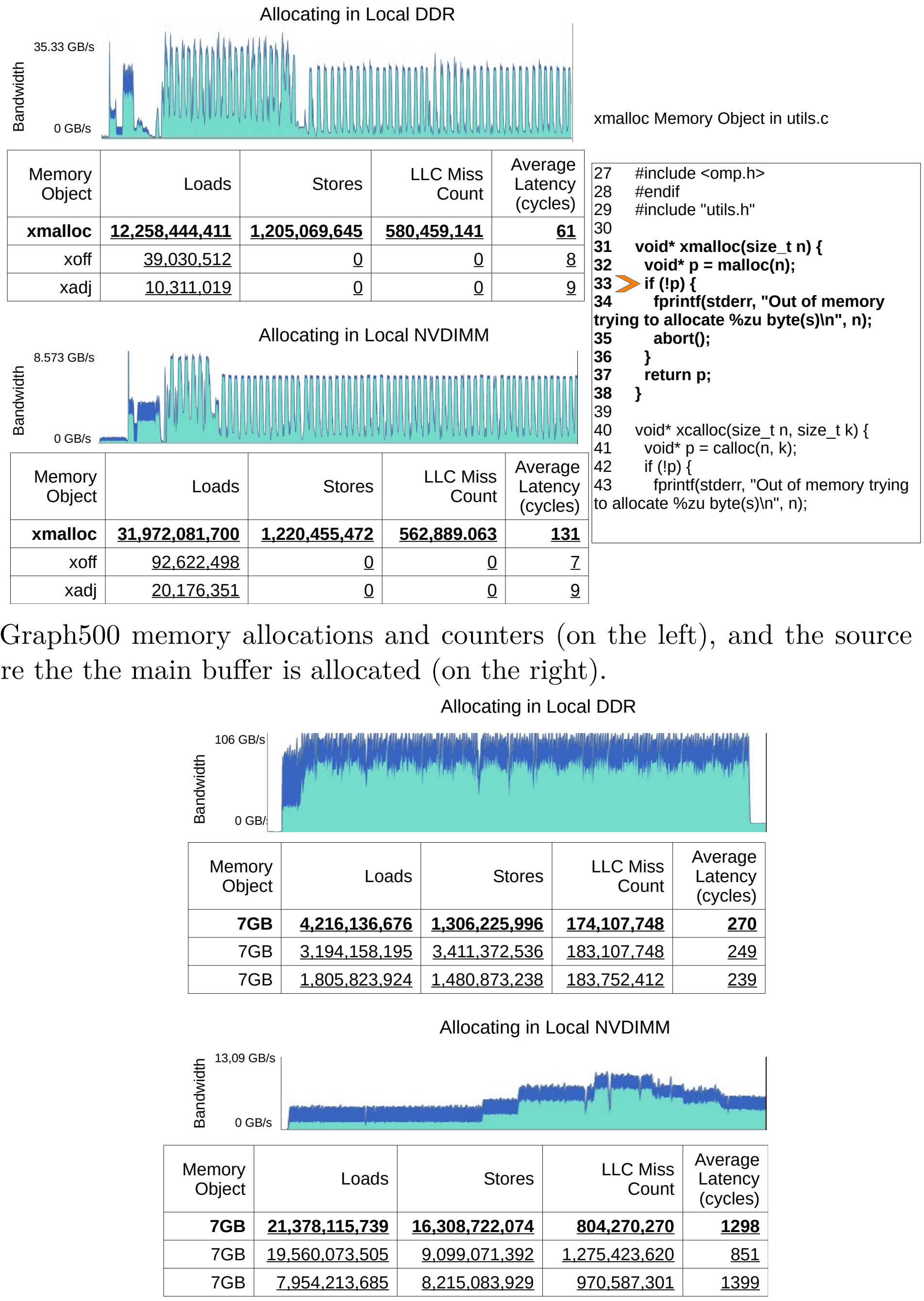
Second, to identify the sensitive data buffers, the memory access analysis may provide details as shown in Figure 4.5a: which kind of memory is being used, the used bandwidth, and the list of buffers ordered by importance. Additionally, we can identify the corresponding source code touching these buffers. It is clear from this figure that the relevant buffer is allocated in `xmalloc` on line 32 (callstacks may also be displayed). LLC Miss Count is important here because it is the last and longest-latency in the memory hierarchy before main memory, meaning that this latency cannot be avoided.

Thanks to this analysis we can now modify Graph500 to allocate this buffer with the latency attribute in our heterogeneous allocator, and get the same optimised performance on different platforms.

We performed the same analysis with STREAM-Triad in Table 4.5 and Figure 4.5b. VTune shows the indicator flag on the parameter *DRAM Bandwidth Bound* or *PMem Bandwidth Bound* parameter depending on where memory is allocated. This shows that the overall application is rather sensitive to bandwidth, as expected. The in-depth analysis of important buffers may then be performed as earlier.

4.3.3 Summary

Benchmarking and profiling are two methods that open the black box of memory attributes which applications can be sensitive to. Benchmarking may easily provides a general idea of the sensitivity of the application. Profiling requires more time for an in-depth analysis, but modern tools are able to give



Memory Object	Loads	Stores	LLC Miss Count	Average Latency (cycles)
xmalloc	**12,258,444,411**	**1,205,069,645**	**580,459,141**	**61**
xoff	39,030,512	0	0	8
xadj	10,311,019	0	0	9

Memory Object	Loads	Stores	LLC Miss Count	Average Latency (cycles)
xmalloc	**31,972,081,700**	**1,220,455,472**	**562,889.063**	**131**
xoff	92,622,498	0	0	7
xadj	20,176,351	0	0	9

xmalloc Memory Object in utils.c

```
#include <omp.h>
#endif
#include "utils.h"

void* xmalloc(size_t n) {
  void* p = malloc(n);
  if (!p) {
    fprintf(stderr, "Out of memory
trying to allocate %zu byte(s)\n", n);
    abort();
  }
  return p;
}

void* xcalloc(size_t n, size_t k) {
  void* p = calloc(n, k);
  if (!p) {
    fprintf(stderr, "Out of memory trying
to allocate %zu byte(s)\n", n);
```

(a) Graph500 memory allocations and counters (on the left), and the source code where the the main buffer is allocated (on the right).

Memory Object	Loads	Stores	LLC Miss Count	Average Latency (cycles)
7GB	**4,216,136,676**	**1,306,225,996**	**174,107,748**	**270**
7GB	3,194,158,195	3,411,372,536	183,107,748	249
7GB	1,805,823,924	1,480,873,238	183,752,412	239

Memory Object	Loads	Stores	LLC Miss Count	Average Latency (cycles)
7GB	**21,378,115,739**	**16,308,722,074**	**804,270,270**	**1298**
7GB	19,560,073,505	9,099,071,392	1,275,423,620	851
7GB	7,954,213,685	8,215,083,929	970,587,301	1399

(b) STREAM-Triad memory objects and counters.

Figure 4.5: Extracts of the *Memory Access* graphic interface in the Intel VTune Profiler. Execution with memory in DRAM (top) is compared to NVDIMM (bottom). The read bandwidth is represented in turquoise, while the write bandwidth is in blue (aggregated on top of read).

Table 4.5: Extracts from the VTune Profiler execution summary for Graph500 and STREAM-Triad using DRAM or NVDIMM.

Application	Target	DRAM Bound in % of Clockticks	PMem Bound in % of Clockticks	DRAM Bandwidth Bound in % of Elapsed Time	PMem Bandwidth Bound in % of Elapsed Time
Graph500	DRAM	29.0%	0.0%	0.0%	0.0%
Graph500	NVDIMM	63.0%	60.9%	0.0%	0.0%
STREAM-Triad	DRAM	63.3%	0.0%	80.4%	0.0%
STREAM-Triad	NVDIMM	43.7%	17.0%	0.3%	2.1%

very useful information. Although this information still requires human intervention, it allows you to determine which buffers are really important for performance and also to determine their sensitivity. This is a critical step towards applying the proper allocation criteria, either using our heterogeneous allocator or in the runtime system.

We believe that our approach brings higher productivity since this sensitivity information can be passed to allocators in a portable manner without having to hardwire information about existing memory kinds into the application code.

Chapter 5

Software Tools for the development on Heterogeneous Memory

Developers must prepare applications and runtime to work on the various heterogeneous memory systems, but it is clear that they not always have access to machines with different memory scenarios. This is why the necessary tools must be provided, so that developers can prepare their applications to face different heterogeneous memory systems. In fact, developers can easily work with the very well known KNL heterogeneous memory system because the configuration remains always the same. However, this changes with NVDIMMs in generic platforms, where the number or kinds of memories can change significantly. Hence, there is a need to test runtime/applications on a wide variety of hardware configurations to make sure our software is portable.

In this chapter, we present the problem of how to prepare applications to support different heterogeneous memory systems without actually having them. First, we have addressed those scenarios, which the developer needs to test the performance of either their already developed application or the runtime in a specific heterogeneous memory scenario. And second, we have addressed different tools, which we can expose memory configurations to applications that do not correspond precisely to the physical environment, i.e., we present some tools that can be useful to emulate heterogeneous memory systems. In fact, this does not expose a true heterogeneous performance, but in this case, the point is to expose the heterogeneous memory system as a real system would do it.

5.1 Performance Simulation

When speaking of performance simulation, we refer to the fact of simulating the behaviour that a certain application should have when it faces a memory scenario that is very possibly difficult for the developer to access. E.g., the case of a future coming system with a different heterogeneous memory system.

As shown in Chapter 3, the analysis of the memory characteristics of different platforms presents the following ordering in terms of bandwidth and latency for Leonide:

$$DRAM_{BW} > NVDIMM_{BW} \tag{5.1}$$

$$DRAM_{Lat} > NVDIMM_{Lat} \tag{5.2}$$

Basically meaning that the best memory in terms of bandwidth is also the best in terms of latency.

For Kona01 this situation changes a little, in where the best memory in terms of bandwidth has no difference with the other in terms of latency (or at least the difference is minimal):

$$HBM_{BW} > DRAM_{BW} \tag{5.3}$$

$$DRAM_{Lat} \simeq HBM_{Lat} \tag{5.4}$$

However, in the previous study we have not in our real platforms a situation where **the best memory in terms of bandwidth is the worst in terms of latency**.

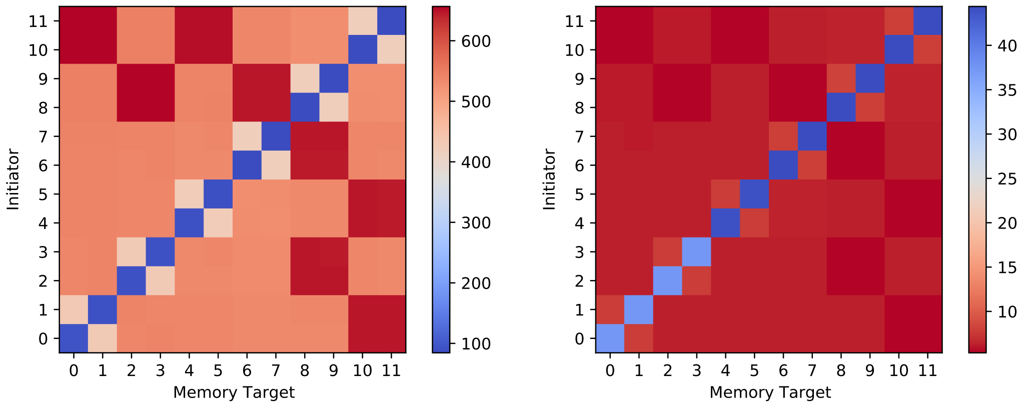
$$Memory(A)_{BW} > Memory(B)_{BW} \tag{5.5}$$

$$Memory(A)_{Lat} < Memory(B)_{Lat} \tag{5.6}$$

Due to this, we have imagined an experimental platform where the architecture allows to have a heterogeneous memory system with the previous characteristics in whereby using some performance simulations techniques predict the behaviour of applications.

5.1.1 NUMA Distance for injecting latency

Non-uniform memory access (NUMA) has been generalised in multi-socket machines, which each socket has an associated memory controller. Due to the relative distance between processors to memory devices, the memory access is non-uniform. Since the birth of the NUMA systems, there have been various investigations that seek to minimise the impact of this situation [2], [97]. However, for the current purpose of simulating an experimental memory environment, these non-uniform accesses can be considered as a possible tool that allows to approach a heterogeneous memory system that we currently do not have at hand.



(a) Latency (in ns).

(b) Bandwidth (in GB/s).

Figure 5.1: Performance heat maps on Souris.

This allows to inject latency through the use of memories in more distant nodes. In Figure 5.1a, we can observe the latency output of mlc on Souris platform (more details in Appendix A.3). However, if analysing the corresponding bandwidth heat-map in Figure 5.1b, we can observe that there is no case where the best memory in terms of bandwidth is the worst in terms of latency, thus, this approach is not enough. However, it is useful in scenarios where the developer seeks to have a greater latency difference regardless of what happens to the other attributes.

In fact, with this method allows to simulate a system where it is useful to have similar bandwidths (between non-local nodes) but different latencies. Non-local nodes could act as different memory kinds depending on their distance from the local CPUs.

$$Memory(A)_{BW} < Memory(B)_{BW} \tag{5.7}$$

$$Memory(A)_{Lat} > Memory(B)_{Lat} \tag{5.8}$$

5.1.2 Bandwidth Throttling

The Intel Resource Director Technology (Intel RDT) is a tool that provides monitoring on cache and memory allocation. It is exposed as a user interface for resource control by Linux Kernel. Intel RDT is able to partition some resources such as the cache hierarchy and for our particular interest the bandwidth [73]. This process is accomplished through the bandwidth throttling mechanism provided by some processors [44]. Since Linux 4.21, resctrl is supported in AMD platforms as quality of service extension (AMD QoS) [8] providing similar characteristics as Intel RDT [1]. The ARM Memory Partitioning and Monitoring (MPAM) is the analogue support to Intel and AMD versions, however, its main features are still under development [2].

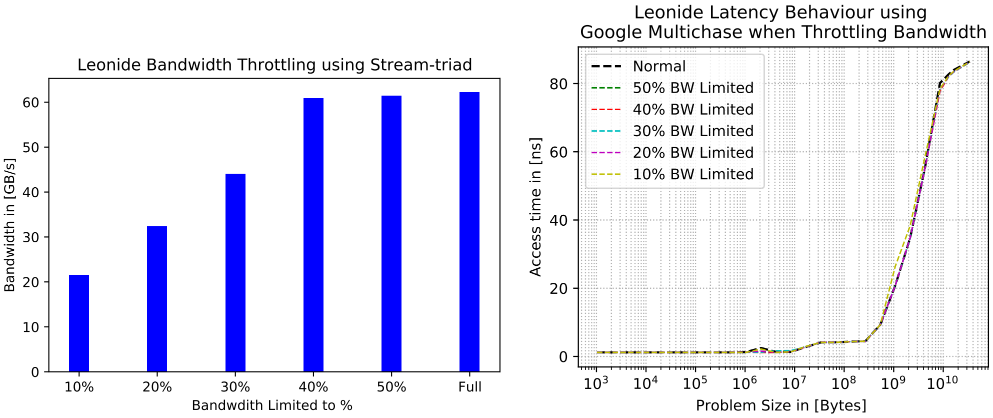
The functionality of partitioning resources such as the bandwidth memory gives an option to change the performance of an application. This allows to add more heterogeneity to a given HMS, and therefore have a tool that allows to simulate the performance of an application by limiting memory bandwidth.

We have tested this functionality on Leonide with STREAM-Triad application. Figure 5.2a shows how resctrl mechanism throttles the bandwidth of DRAM. Although, the memory bandwidth throttling effect could be observed, it does not match the requested throttling percentage. This is why, a significant change is only observed after putting the resctrl at 40%. It is due, because throttling applies to memory requests in the CPU but DRAM is slower than the maximum memory access performance of the CPU.

In Figure 5.2b, we can see several overlapped lines, meaning that latency is not affected (or that the affection is negligible).

[1] https://www.phoronix.com/scan.php?page=news_item&px=AMD-QoS-Landing-Linux-4.21

[2] https://static.linaro.org/connect/lvc20/presentations/LVC20-108-0.pdf



(a) STREAM-Triad Bandwidth output.

(b) Latency output-report of Google Multichase.

Figure 5.2: Throttling DRAM memory on Leonide.

This approach allows to change the heterogeneity of a memory system in terms of bandwidth without affecting latency. In the case of Leonide we have affected the memory system creating a DRAM' with less bandwidth than the original DRAM; and maintaining the latency values. All these generates the following memory system:

$$DRAM_{BW} > DRAM'_{BW} > NVDIMM_{BW} \tag{5.9}$$

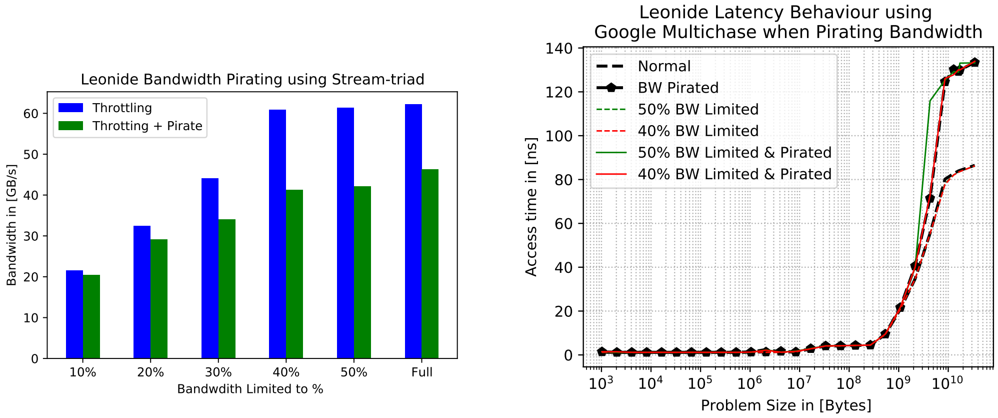
$$DRAM_{Lat} = DRAM'_{Lat} > NVDIMM_{Lat} \tag{5.10}$$

In fact, this case could be used to simulate a platforms such as our fictitious 3MK system described in Chapter 3 in where DRAM could act as HBM and DRAM' as DRAM.

5.1.3 Pirate Bandwidth

Memory pirating is a performance simulation method that allows to control how much memory is available to the application by co-running it with a memory-"stealing" application [27]. Indeed, the pirate steals memory bandwidth from the application by ensuring that its entire workload always bound on the memory target. This effectively reduces the memory bandwidth available to the application. Therefore, it is possible to simulate the performance of an application, simulating the behaviour of a memory by using the loss of performance generated by the exchange of hardware resources due to the simultaneous execution of several applications on the same compute node.

This method in the same manner as the bandwidth throttling aims to affect the performance in terms of bandwidth. In Figure 5.3a, we can observe that in full bandwidth, the pirate effect affects the bandwidth even if it is combined with bandwidth throttling. However, the pirate in contrast to the throttling affects latency as seen in Figure 5.3b.



(a) STREAM-Triad Bandwidth output.

(b) Latency output-report of Google Multichase.

Figure 5.3: Throttling and/or pirating local DRAM memory bandwidth on Leonide.

The pirate compared to the throttler, does not depend on how it has been implemented in AMD, Intel or ARM. It simply tries to make bandwidth noise to affect the performance of an application. This allows to obtain another strategy to be able to affect the heterogeneity of a memory system. However, this process affects not only bandwidth, but also affects latency. So it might not be a useful technique in some situations. E.g. the resulting heterogeneity changes Leonide in this way:

$$DRAM_{BW} > DRAM'_{BW} > NVDIMM_{BW} \tag{5.11}$$

$$DRAM'_{Lat} > DRAM_{Lat} > NVDIMM_{Lat} \tag{5.12}$$

One drawback of using the pirate is that it is not easy to control. For example, it is difficult to tell the pirate to use a certain amount of bandwidth. And on the other hand, its impact on applications depends on how much bandwidth these applications actually use.

5.1.4 Summary of Performance Simulation

This section has presented three techniques that allow an application to simulate different behaviours. The bandwidth throttling by means of resctrl, is the only one of these techniques that is natively supported, allowing to limit the bandwidth of an application without affecting the latency. The pirate, which also seeks to limit the bandwidth of an application, although it may affect its latency. And, the NUMA distance latency injection case that is an artifact that allows users who need to test their applications with higher latencies.

The use of these techniques requires more advanced criteria from the developer when their objective is to affect only one of the memory metrics. Since he/she will have to make certain adjustments, and also the need to combine these techniques. Despite this, these techniques allow creating fictitious platforms with different HMS than what is available in real hardware platforms.

5.2 Environment Emulation

The emulation of a memory system allows developers to prepare their applications for next-generation memory systems. Emulating these systems consists of seeking somehow to expose memory devices that are not physically located in the nodes as if they were present. In emulation we cannot change the performance as in the previous section. It usage is mainly dedicated to expose different HMS systems to prepare or adapt applications to access different kinds of memory. I.e., test the heuristics and algorithms developed to select the appropriate target for each buffer.

Emulation can be done using several options and we have classified them in hardware emulation, OS emulation, and software emulation. In Figure 5.4 we are able to see an overview of the different techniques to presented in the following subsections.

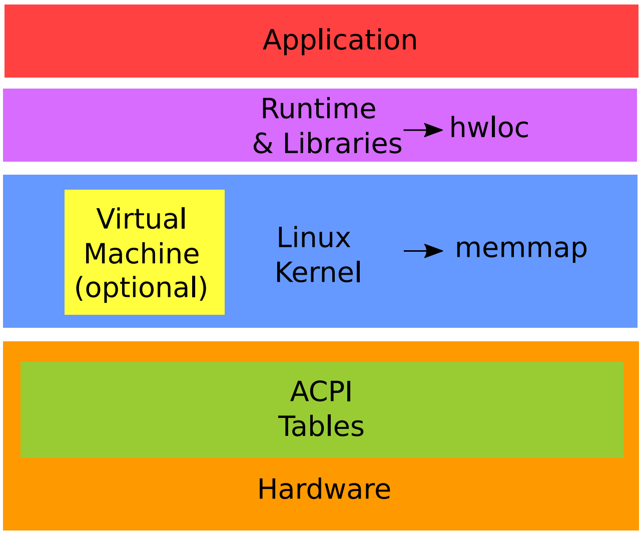


Figure 5.4: Emulation software stack.

5.2.1 Hardware Emulation Level

Emulating the HMS through hardware refers to the use of a real hardware device to mimic the function of another HMS. The emulation can be done in different manners, but all of them are designed to behave like an entirely different hardware platform than the one it runs on. It allows to debug and verify applications, the runtime system, or a new system under design.

5.2.1.1 Replacing ACPI Tables

Advanced Configuration and Power Interface (ACPI) was specified to establish a common interface for platform-independent configuration. This means that these systems when they are turned on and before the OS is loads, the BIOS places ACPI tables in memory [25]. It helps the operating system to efficiently configure the hardware [102] (in our case the HMS).

Some of this tables were mentioned in Section 3.4. In fact, System Locality Distance Information Table (SLIT), System Resource Affinity Table (SRAT), NVDIMM Firmware Interface Table (NFIT) and Heterogeneous Memory Attributes Table (HMAT) are the main tables related to the memory system.

Therefore the fact of modifying these tables appropriately means a way for emulate memory systems, however, only HMAT would really change heterogeneity in terms of latency and bandwidth.

The actual way to emulate the HMAT table is:

1. Use the HMAT tables located on an available machine that would serve as a base for a memory system to be modified. The current machine tables exposed by Linux in /sys/firmware/acpi/tables

2. Disassemble it with ACPI Source Language compiler/decompiler (iasl).

3. Modify some values or split new proximity domains to create new memory targets (e.g., bandwidth and/or latency).

4. Reassemble it again with iasl.

5. Then, load it into the kernel at boot from initrd, which is a facility to provide early software environment during the boot.

6. The emulation will be complete when these tables are loaded. To carry out this process they can be compiled using iasl [3].

It is also posible to modify NFIT, SLIT and SRAT. It requires to modify the proximity domains that corresponds to memory targets. This task could be not easy, however, daxctl split-acpi allows to simplify this operation by dividing NUMA nodes in two. It is useful when adding fictitious NVDIMMS to

[3] https://www.kernel.org/doc/Documentation/acpi/initrd_table_override.txt

a DRAM-only machine, by spliting DRAM proximity domain into half DRAM and half NVDIMM and then modify values on HMAT.

Figure 5.5 shows the results of using the first HMAT sample provided in https://github.com/rzwisler/hmat_examples. 00 in Data Type means that the entire entries contain latency values. Similarly, 03 meaning bandwidth and 01 meaning read latency. In fact, real values come from multiplying the Entry Base Unit value with the Entry value. E.g., the entry 01F4 (500) multiplied by the entry base unit C8 (128) gives 64000 pico-seconds.

```
                      Signature : "HMAT"    [Heterogeneous Memory
                                                      Attributes Table]
                   Table Length : 00000158
                       Revision : 01
                                .
                                .
                                .
                 Structure Type : 0001 [System Locality Latency
                                             and Bandwidth Information]
                       Reserved : 0000
                         Length : 00000048
          Flags (decoded below) : 00
               Memory Hierarchy : 0
                      Data Type : 00
                      Reserved1 : 0000
  Initiator Proximity Domains # : 00000002
     Target Proximity Domains # : 00000004
                      Reserved2 : 00000000
                Entry Base Unit : 00000000000000c8
Initiator Proximity Domain List : 00000000
Initiator Proximity Domain List : 00000001
   Target Proximity Domain List : 00000000
   Target Proximity Domain List : 00000001
   Target Proximity Domain List : 00000002
   Target Proximity Domain List : 00000003
                          Entry : 01F4
                          Entry : 03E8
                          Entry : 01F4
                          Entry : 03E8
                          Entry : 03E8
                          Entry : 01F4
                          Entry : 03E8
                          Entry : 01F4
```

Figure 5.5: Modifying entries into HMAT.

Emulating a heterogeneous memory system by modifying ACPI tables allows not only to prepare applications. This allows to prepare runtimes to support the arrival of emerging memory systems.

5.2.1.2 Qemu

Another effectively way to emulate HMS is do it within a virtual machine. The virtual machine would not have direct acces to the server hardware. Instead, this emulation level redirects traffic between physical and virtual hardware.

Quick Emulator (QEMU) is a free opensource emulator based on dynamic translation of binaries. I.e., it convert the binary code of the source architecture into understandable code by the host architecture. It is also able to virtualise inside an OS be it Linux, Windows and others.

For our purpose, QEMU allows to create a very flexible configuration of NUMA nodes topology [4]. The command described in Figure 5.6 show us that in QEMU we are able to define numa nodes, change distances values and define NVDIMMs.

```
sudo qemu-system-x86_64 -hda ubuntukernel.img -smp cpus=4
-numa node,cpus=0-1,nodeid=0 -numa node,cpus=2-3,nodeid=1
-numa dist,src=0,dst=1,val=35 -cpu host -m 8G,slots=4,maxmem=32G
-machine pc,accel=kvm,nvdimm=on
-enable-kvm -object memory-backend-file,id=mem1,share=on,
                        mem-path=/dev/dax0.0,size=4G,align=2M
-device nvdimm,id=nvdimm1,memdev=mem1,label-size=2M
```

Figure 5.6: Qemu command line to emulate HMS.

As can be seen in the previous point, one can modify the ACPI tables of a real machine. This can also be done in virtual machines regardless of the virtualiser engine. In this way allowing to combine these two approaches. However, the goal of using virtual machines for our purpose is to expose virtual/different hardware without requiring admin access to modify the initrd.

5.2.2 OS Level

Another manner to emulate heterogeneous memory systems is through operating systems.

In fact, since Linux Kernel v4.0 it is possible to emulate NVDIMMs. Linux provides the memmap kernel option [5], allowing to reserve unasigned memory for emulate NVDIMMs.

[4] https://futurewei-cloud.github.io/ARM-Datacenter/qemu/how-to-configure-qemu-numa-nodes/

[5] https://www.kernel.org/doc/Documentation/admin-guide/kernel-parameters.txt

The memmap feature should be passed on the kernel command line (e.g., in GRUB) in this manner `memmap=nn[KMG]!ss[KMG]`, where nn is the size of the region to reserve, and ss is the offset.

The following paragraph, shows the method to emulate on debian-based operating systems [6]:

– New memory mapping of 5GB starting at the 11GB boundary (i.e., from 11GB to 16GB).

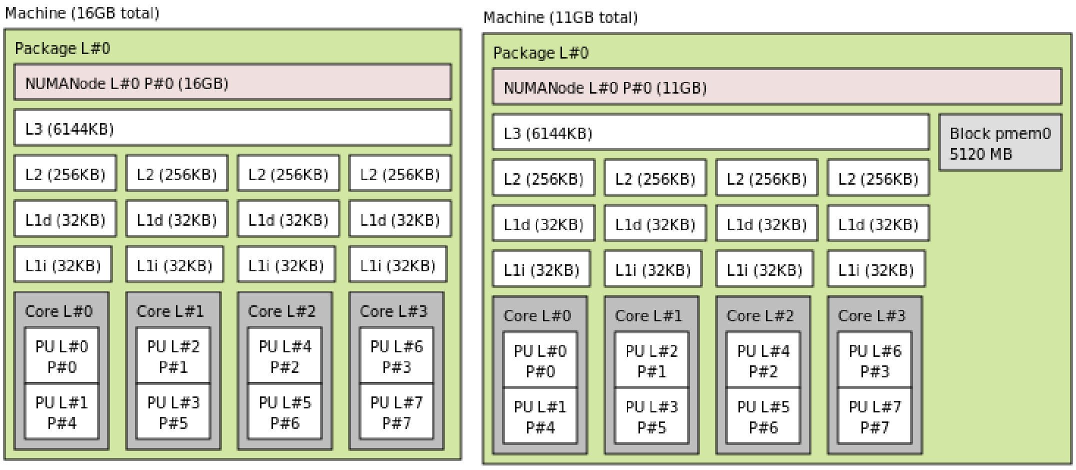
```
$ sudo vi /etc/default/grub
```

– Add or edit the "GRUB_CMDLINE_LINUX" entry to include the mapping.

```
GRUB_CMDLINE_LINUX="memmap=5G11G"
```

– Then update grub and reboot.

```
$ sudo update-grub2
```



(a) Original lstopo output.

(b) `memmap=5G!11G` lstopo output. 5GB disappeared in the NUMA node and they appeared as 5GB of persistent memory `pmem0`.

Figure 5.7: Topology of a single node computer with 16GB physical DRAM.

Using this kernel tool gives a great facility to developers access to NVDIMMs, It allows to use fictitious NVDIMMs in App Direct mode or in System-RAM mode. When using System-RAM mode memmap puts the

[6] https://docs.pmem.io/persistent-memory/getting-started-guide/creating-development-environments/linux-environments/linux-memmap

memory in node 0 instead of creating a separate node 1, so it could be useless for our purpose. For App Direct mode, memmmap is effectively used for storage, which allows to have a fictitious HMS by mapping files instead of just allocating in NUMA nodes. Figure 5.7 represents the topology before and after adding memmap option on a single Laptop Computer with a debian-based OS and 16GB of DRAM.

This method allows to emulate NVDIMMS given a DRAM. The oposite, emulating DRAM given an NVDIMM is an official feature, and is presented in the next chapter when describing NVDIMM System-RAM mode and 2-Level-Memory mode.

5.2.3 Software Level

Since most HPC runtimes and applications can read hardware topology, exposing different topology will change their behaviour. As seen before, we could do it through modifying the ACPI tables or modify options on the kernel, however, this is rescricted to administrator users and using virtual machines is not alway convenient. Fortunately, hwloc is often used as an intermediate layer between hardware and topology discovery in HPC, hence we can use hwloc to "lie" to applications

When we talk about emulating HMS through software layer, we mainly refer to the use of hwloc to expose the memory system.

hwloc main functionallity allows to obtain a hierachy of the main computing elements, such as: NUMA nodes, Caches, Processor Sockets, Cores and Threads. Through lstopo tool, we can expose the topology provided by hwloc in different formats. Specifically, lstopo supports properly xml extension. It permits reuse this file later and do it on another machine if wanted.

This is not just a visual change of the topology. This allows developers to prepare applications taking into account in advance possible scenarios where heterogeneity and locality may vary.

hwloc can be used in three cases:

1. Creating a topology through a synthetic description. E.g., Figure 5.9, shows the topology of using the following command-line lstopo –input "Package:2 L3:1 Group:2 [numa(memory=1GB)] [numa(memory=1TB)] L2:16 L1:1 Core:1 PU:2".

2. Modifying the HMS topology throught an xml file. In fact, we have already presented an example of this with the 3MK machine in Chapter 3. Figure 5.10 shows the process about generating our fictitious 3MK machine. This process permits to emulate more than one memory kind local to a CPU.

3. Modify memory attributes and values in the xml as shown in Figure 3.11.

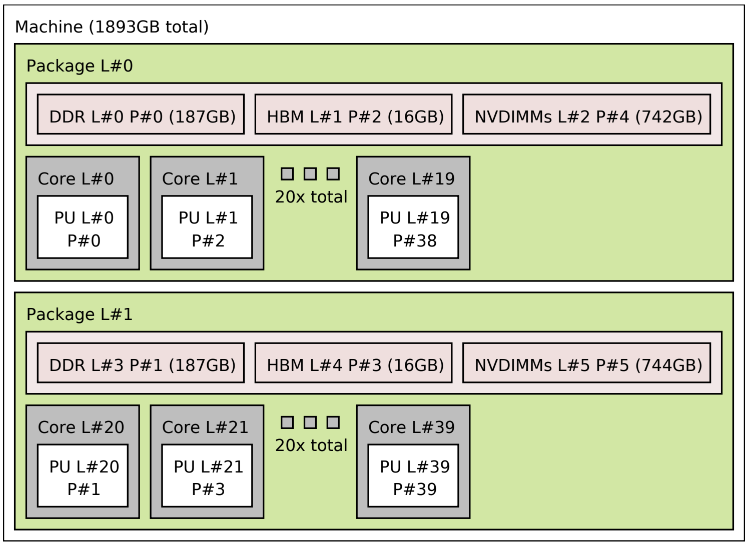


Figure 5.8: lstopo graphical output ot a fictitious 3-memory-kind machine modified from Leonide platform.

5.2.4 Summary of Emulation

The emulation of heterogeneous memory systems as seen could be done in several ways. When using Hardware emulation techniques such as modifying HMAT tables, we are able to add heterogeneity just by modifying attribute values, being able to emulate HBM, DRAM and NVDIMMs. The case of QEMU allows to configure DRAM and NVDIMMs. To emulate HBM with QEMU, the way should consist in modify ACPI tables.

Performing the emulation of a HMS through the kernel command-line option memmap allows to have an heterogeneous memory system with DRAM and NVDIMMs, in fact, part of DRAM is used to emulate NVDIMM in App Direct mode with a mmaped file or as an additional NUMA node, details of this are presented in Chapter 6.

Software level emulation help specially in cases where developers have not fully administrative permits to manage the real hardware. Through hwloc, developers are capable to emulate NVDIMMs, DRAM, HBM and other memory devices. Thankfully, hwloc is being used oftenly as a middleware between hardware and topology discovery. I.e., hwloc is widely spread among HPC application developers, so its use is not unknown.

Both the simulation of the performance of a possible HMS application and the emulation of the topology of a certain HMS, allowing developers to give their applications development greater flexibility and portability. That is, developers could make their applications more and more heterogeneity-aware of the memory system.

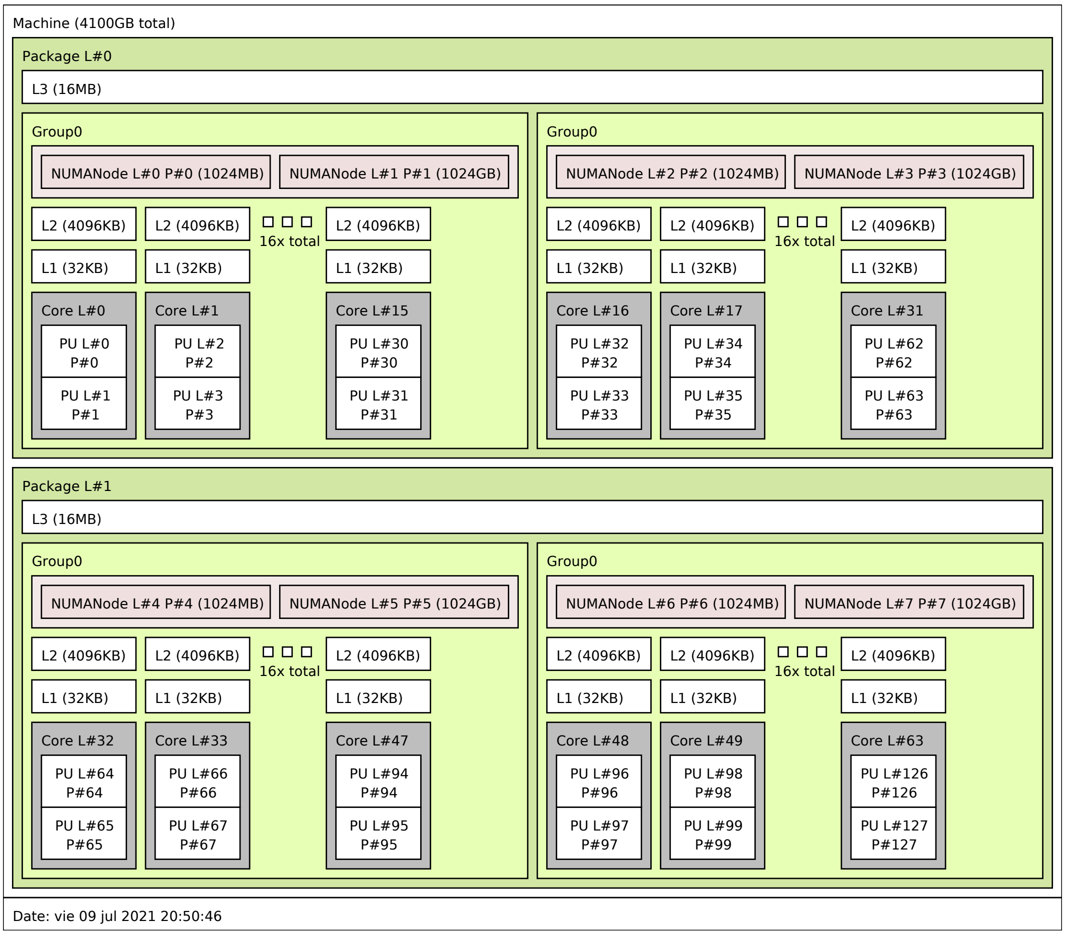
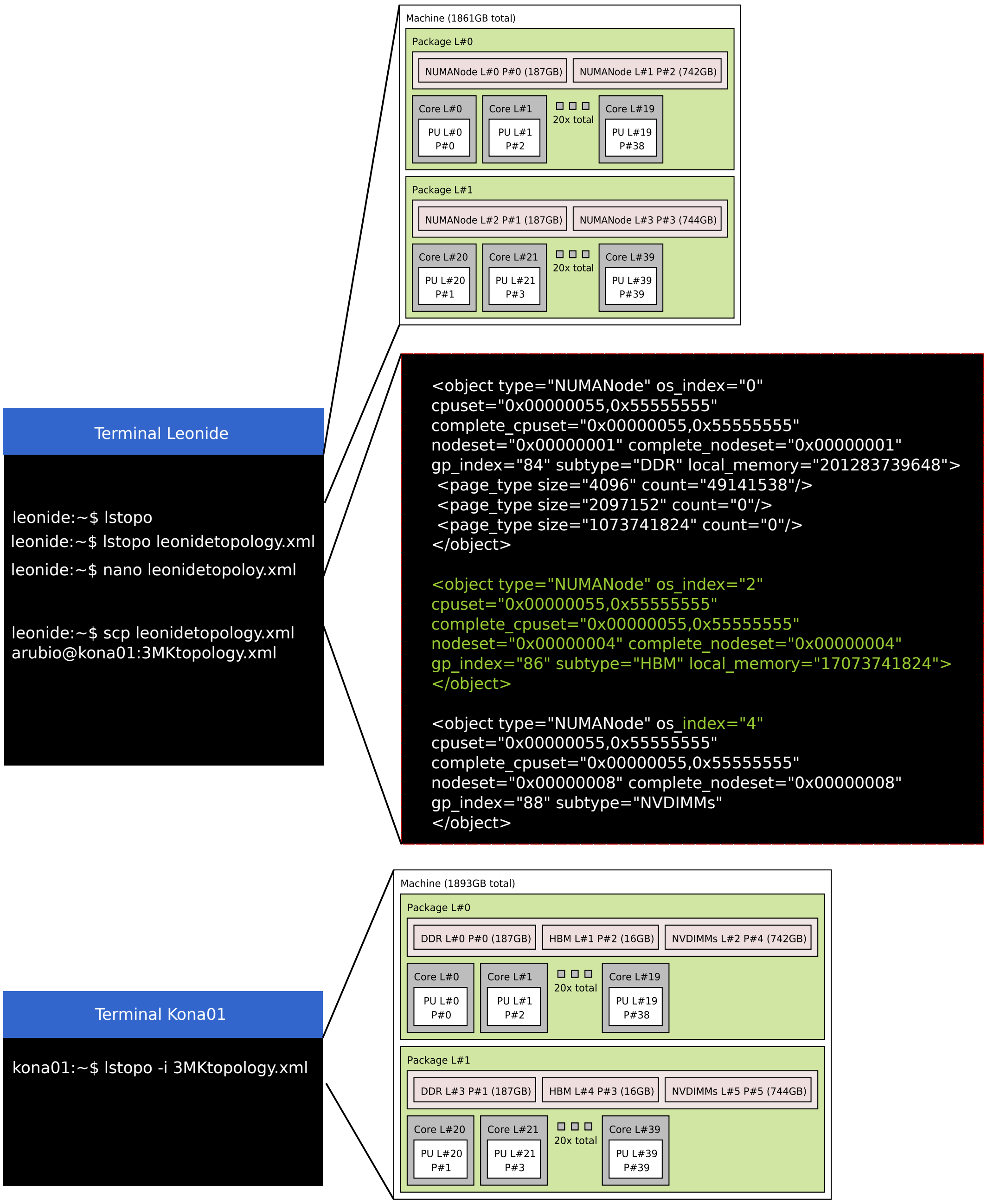


Figure 5.9: lstopo graphical output using the synthetic description `Package:2 L3:1 Group:2 [numa(memory=1GB)] [numa(memory=1TB)] L2:16 L1:1 Core:1 PU:2`.



```
leonide:~$ lstopo
leonide:~$ lstopo leonidetopology.xml
leonide:~$ nano leonidetopoloy.xml

leonide:~$ scp leonidetopology.xml
arubio@kona01:3MKtopology.xml
```

```
<object type="NUMANode" os_index="0"
cpuset="0x00000055,0x55555555"
complete_cpuset="0x00000055,0x55555555"
nodeset="0x00000001" complete_nodeset="0x00000001"
gp_index="84" subtype="DDR" local_memory="201283739648">
 <page_type size="4096" count="49141538"/>
 <page_type size="2097152" count="0"/>
 <page_type size="1073741824" count="0"/>
</object>

<object type="NUMANode" os_index="2"
cpuset="0x00000055,0x55555555"
complete_cpuset="0x00000055,0x55555555"
nodeset="0x00000004" complete_nodeset="0x00000004"
gp_index="86" subtype="HBM" local_memory="17073741824">
</object>

<object type="NUMANode" os_index="4"
cpuset="0x00000055,0x55555555"
complete_cpuset="0x00000055,0x55555555"
nodeset="0x00000008" complete_nodeset="0x00000008"
gp_index="88" subtype="NVDIMMs"
</object>
```

```
kona01:~$ lstopo -i 3MKtopology.xml
```

Figure 5.10: Emulation of the 3MK platform from Leonide. And using it in Kona01 (in green the modification).

Chapter 6

Management of Heterogeneous Memory in Batch Schedulers

Computing nodes are increasing complex, with tens of cores. Co-scheduling allows to optimise the utilisation of these nodes in where applications are able to use distinct hardware resources to avoid the underutilisation of available nodes [81]. Co-scheduling multiple jobs on such nodes, is a useful strategy for making sure all powered-on cores are used in HPC centers. However, sharing nodes between multiple jobs also comes with issues such as: contention in the memory subsystem or cache pollution. Resource partitioning is an interesting way to avoid such issues thanks to operating system features such as Linux Cgroups.

The emergence of non-volatile memory DIMMs brings new strategies for the management of data in HPC applications. In fact, they support multiple hardware and software configurations spanning from huge volatile capacities to high-performance storage, which can be used as burst buffers or for recovery after fault.

We focus in this chapter on the co-scheduling of jobs with different needs, and on the partitioning of these new hardware resources between them. We compare the possible hardware configurations and advocate for the use of the 1-Level-Memory mode with namespaces and explicit NUMA memory management.

The rest of this chapter is organised as follows. First, we present KNL experiments with memory modes and discuss some of its possible configurations, and how it afforded co-scheduling. Then, we present NVDIMMs hardware and its possible hardware and software configurations. Afterwards, co-scheduling jobs with different requirements is discussed before we explain how to partition resources between them. Finally, a discussion and summary section is presented.

6.1 Managing the different KNL configuration modes

Although this chapter will focus on the opportunities that NVDIMMs bring, it is timely to briefly explain how KNL was used in terms of scheduling of jobs and the management of the memory system with the different configurations.

6.1.1 KNL configurations

As presented in Section 2.3.1.1, KNL has three main memory modes: Flat, Cache, and Hybrid that can be configured in the BIOS.

In Flat mode as Figure 6.1 shows, the user sees the HBM exposed by the operating system as an extra NUMA node. For the system, NUMA node 0 corresponds to DDR memory and NUMA node 1 is the on-package HBM. To enable this mode, the KNL server needs to be rebooted. The advantage of this mode is that HBM acts as addressable memory, where developers have more control. However, this mode is not transparent and needs application rework from developers.

In Cache mode as Figure 6.2 shows, HBM becomes transparent to the OS and is managed by the hardware as a large cache in front of DRAM. The main advantage of this memory mode is that is managed by the platform transparently to software, i.e., developers do not need to modify their applications. However, one has to hope its memory access will be properly managed by the hardware cache.

Finally, Hybrid mode as Figure 6.3 shows, partitions the HBM between Flat and Cache mode separately.

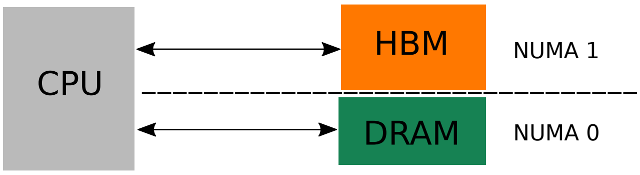


Figure 6.1: HBM configured in Flat mode.

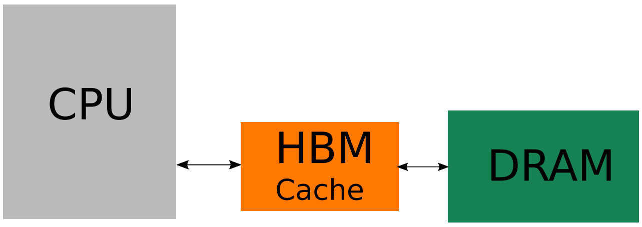


Figure 6.2: HBM configured in Cache mode.

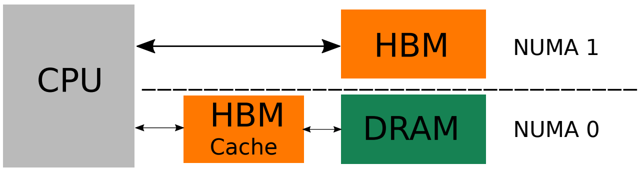


Figure 6.3: HBM configured in Hybrid mode.

There is also another setup in the BIOS that allows to have three different clustering modes: Quadrant Mode, Sub-NUMA Clustering Mode, and All-to-All mode. These modes allow splitting the processor into different sets of cores for managing Cache coherence and/or NUMA affinity.

The All-to-all mode allows having an unbalanced memory distribution across two DRAM memory devices. It does not implements affinity between tiles and memory uniformly. I.e., it hashes all memory address across distributed directories.

The Quadrant mode divides the chip in four quadrants. It permits to have affinity between distributed directories and the memory. However, this is transparent to applications which only see one set of 64 cores.

Sub-Numa Clustering mode (SNC) exposes quadrants as individual NUMA domains to applications that can profit from lower latencies, reducing the use of remote quadrant memory.

6.1.1.1 Workload Manager

The 3×3 KNL modes presented above work well depending on the application. For instance, SNC is good for applications that already implement NUMA locality efficiently. Quadrant mode works for applications where the entire dataset is used by all threads. All the presented modes could be requested by the user using the batch scheduler. Hence, HPC centers need to provide access to KNL in different modes. However, there is a big disadvantage in changing from one mode to another, it could imply that some or all nodes reserved may be rebooted, and this process could add a delay of about 20 minutes before the job starts execution [1].

KNL had many possible configurations that were very well known in advance while NVDIMMs are much more flexible, hence, the same issue appear with NVDIMMs in data centers.

6.2 Managing NVDIMM configuration modes

As mentioned in Chapter 2 Non-volatile memory DIMMs have been available for several years, as DRAM DIMMs with a battery to save data to a flash backup on power loss. However, software support was not ready until recently. These memory DIMMs are inserted in usual memory slots just like normal DIMMs (DRAM) as in Leonide in Figure 6.4.

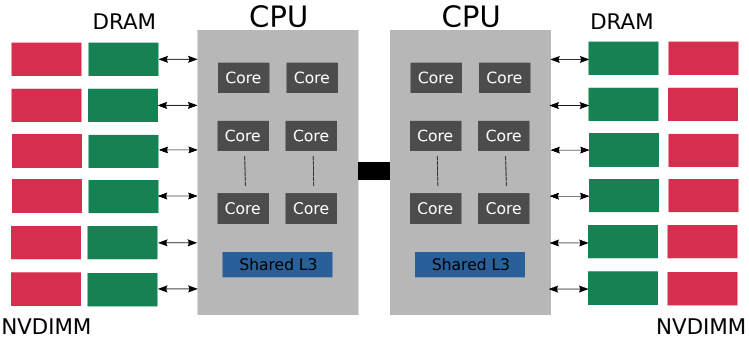


Figure 6.4: Leonide Platform: Dual-socket Xeon platform with 6 channels per processor, with one Optane DC Persistent Memory Modules (DCPMM) and one DRAM each.

In previous Chapters, **NVDIMMs were used in a volatile memory mode that permits use them as extra NUMA nodes**. However, in this chapter a step back is done to present other configuration modes.

Intel NVDIMMs can be configured as individual Regions or as Interleaved Regions. Interleaving implies that all the entire region data is lost whenever

[1] https://www.nersc.gov/assets/Uploads/Using-KNL-Processors-Feb2019.pdf

a single NVDIMM fails. However, interleaving is still expected to be used by default because it increases the memory bandwidth by using multiple channels simultaneously. Non-interleaved regions are expected to be useful for separating small and independent jobs such as virtual machines.

In addition to regions, Xeon processors are capable to use these NVDIMMs as normal (volatile) memory [9]. To do this, NVDIMMs have two operating modes: Memory mode and App Direct mode. This configuration is done in the BIOS or through a tool called `ipmctl` and allows to have the entire NVDIMMs in one mode or partition them between both. A reboot is required for enabling the new configuration.

6.2.1 Memory Mode and 2-Level-Mode

Intel Xeon processors can be configured in 2-Level-Memory mode (2LM). This exposes the Memory Mode part of NVDIMMs as volatile memory an uses the DRAM as a memory-side cache in front of it, as shown in Figure 6.5. This is similar to the KNL Cache mode. It provide a huge capacity of volatile memory (terbaytes) with good performance as long as the memory access hit the cache in DRAM (hundreds of gigabytes).

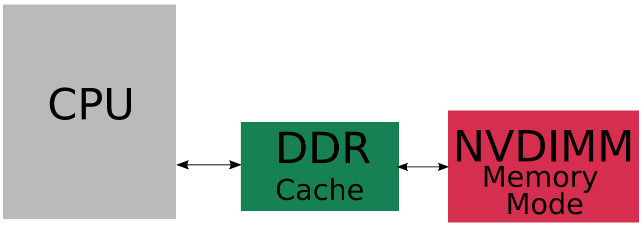


Figure 6.5: 2-Level-Memory mode (2LM) uses DRAM as a Memory-side Cache in front of the Memory Mode part of NVDIMMs exposed as normal volatile memory.

6.2.2 App Direct and DAX and 1-Level Memory for storage

Intel Xeon processors can also be configured in 1-Level-Memory mode (1LM), where DRAM is put as the main volatile memory as seen in Figure 6.6. Here, the App Direct part of NVDIMMs is exposed as persistent memory regions that can be used as a disk (e.g., /dev/pmem0).

This Linux kernel provides Direct Access (DAX) implementations to avoid the need for intermediate copy and page-cache allocations. This enables the mapping of the actual backend data directly in the application virtual memory and permits the use of loads and stores.

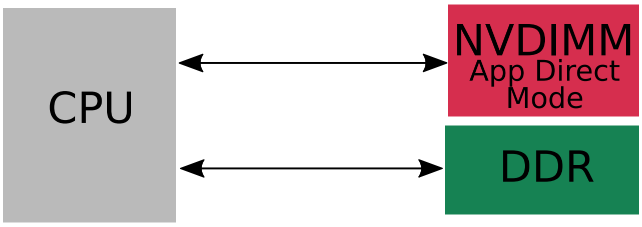


Figure 6.6: 1-Level-Memory mode (1LM) using App Direct mode uses DRAM as the main memory, while NVDIMMs are exposed as a persistent memory region that is usually used as storage.

The file-system mode (FSDAX) is the default Linux mode that is used in most cases because it exposes persistent storage as a normal file system. Applications may use these files as usual. However, optimal performance requires the application to be modified for DAX: they should stop using explicit file access (read/write requires a copy) and rather map files to virtual memory instead (to directly access data in NVDIMMs).

If a file-system is not desired, Linux also provides the Device DAX mode that exposes NVDIMMs as a large mmap'able linear space where applications may manually store their datasets. It was designed to expose large regions of non-volatile memories to specific applications like virtual machines, but it is actually much more useful than this. Device DAX requires significant rework of applications because they have to manually separate independent data without the help of independent files in a file-system. However, as already explained above, DAX requires applications to be rewritten to take advantage of improved performance (map files instead of read/write). Therefore, we believe additional application changes for supporting Device DAX are not a significant obstacle.

6.2.3 System-RAM mode

Developers choose between 2LM and 1LM configurations presented in previous sections depending on whether they want a large capacity volatile memory or storage. Both cases use NVDIMMs, but the 2LM mode has higher bandwidth thanks to the DRAM cache [41, 103, 39]. However, 1LM latency is better than 2LM in the case of cache-miss because there is no need to look up the data in the DRAM cache first [56].

We now present an additional mode enabled in software that provides even more flexibility. Linux kernel 5.1 brought a new Device DAX option: the System-RAM mode (kmem DAX kernel driver). It replaces the persistent memory region exposed as storage with additional NUMA nodes. Applications can allocate memory in these nodes as on any NUMA platform [35], as seen in Figure 6.7. Thus, they see one NUMA node per kind of memory, so they create a 2MK platform as used in previous chapters. This may be considered similar to Intel Xeon Phi *Flat* mode presented in the previous section, where both fast and slow memories are exposed as separate NUMA nodes.[2]

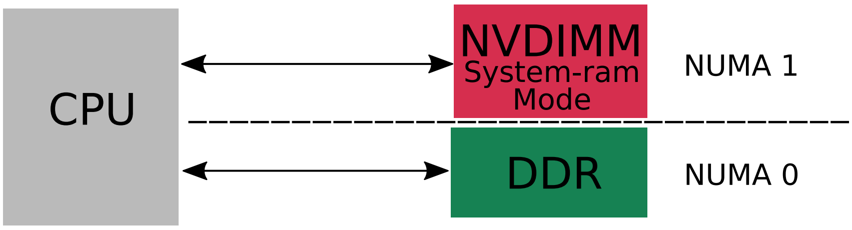


Figure 6.7: 1-Level-Memory mode (1LM) using System-RAM mode allows appear the NVDIMM as an additional NUMA node apart of DRAM.

It means, that applications now have to manually allocate in one of the nodes, depending on the required performance for each dataset. This requires more work from application developers, but provides more flexibility than 2LM. This is the same tradeoff than choosing between the easy-to-use KNL Cache mode and the advanced KNL Flat mode where one has to allocate buffers in the right target. Hence, we believe higher performance is achievable thanks to System-RAM mode if the developer carefully places performance-sensitive data in DRAM (faster than 2LM DRAM-cache [12]) and other large datasets in NVDIMM (slower than 2LM DRAM-cache [41, 103] but not performance-sensitive).

6.3 Co-scheduling jobs with memory and storage needs

Modern HPC nodes feature lots of cores and memory. They are therefore good candidates for co-scheduling several small jobs. Unfortunately, node sharing raises multiple issues in terms of performance [93]. Hence, we now explain how to partition nodes equipped with NVDIMMs.

6.3.1 Hardware Partitioning in 2LM

We explained in the previous section that partitioning NVDIMMs between different modes is possible. However, the hardware configuration has to match the job requirements. We now look at the case where some jobs want a 2LM configuration (Memory Mode for large amounts of volatile memory) and some

[2] NVDIMMs 1LM and 2LM modes are similar to KNL *Flat* and *Cache* modes. However, NVDIMMs do not enable a KNL-like *Hybrid* mode: KNL could partition the fast memory (MCDRAM) between cache and normal memory. NVDIMMs rather allow partitioning the slow memory between cached (by the fast memory, DRAM) and uncached.

others want 1LM (App Direct for persistent storage). The only way to have both Memory Mode and App Direct available at the same time in a machine is to configure the processors in 2LM.

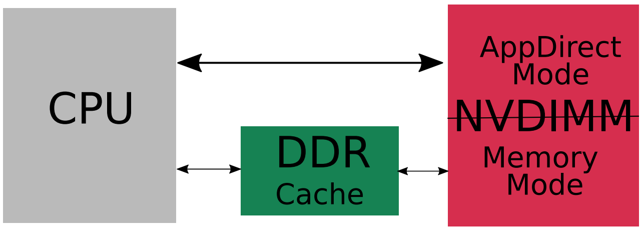


Figure 6.8: 2-Level-Memory enables exposing both Memory Mode as DRAM-cached main memory and App Direct as storage.

However, this configuration has major drawbacks: First, the administrators would have to choose a good ratio for NVDIMM, partitioning between Memory and App Direct mode. This ratio depends on the needs of all jobs that will be scheduled simultaneously on a node, and setting up the ratio requires a reboot of the node.[3]

Secondly, locality issues arise as shown in Figure 6.9: If a processor is allocated to a 1LM job, its local NVDIMMs should be entirely set in App Direct. However, it means there is no local memory: both local DRAM and NVDIMM cannot be used as volatile memory (DRAM is entirely used as a cache; NVDIMMs are entirely used as App Direct). Cores of this socket would therefore use remote DRAM on the second processor, which incurs bad performance.

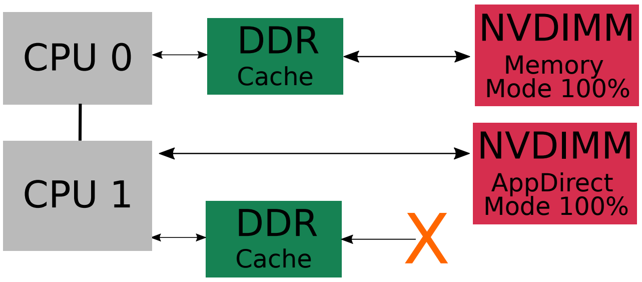


Figure 6.9: Allocating one socket to a job that wants 100% Memory Mode and the other socket to a job that wants 100% App Direct causes the latter to have no local memory, and its DRAM cache is useless.

In the end, we believe using 2LM to share a node with such different jobs is not a good idea, and we do not expect significant improvements in future hardware platforms. Administrators would rather create one set of 1LM nodes

[3] Additionally a 32G granularity seems to constrain possible ratios in hardware.

and a separate set of 2LM nodes, similar to what was shown in Section 6.1.1.1 [4] and possibly reconfigure them as in Section 6.2. However, next subsection shows how 1LM may actually offers a more flexible solution.

6.3.2 Flexible Co-Scheduling with 1LM and System-RAM NUMA nodes

We explained in the previous section that 2LM requirements incur too many drawbacks. Therefore, we propose not to use 2LM anymore. As explained in Section 6.2.3, Device DAX may be exposed as additional NUMA nodes in System-RAM mode. This provides lots of volatile memory that 2LM applications require, but requires developers to explicitly manage allocation between fast and slow memory. We believe that this additional work for developers is a good trade-off because of the flexibility it provides to users and administrators.

Hence, we propose the following strategy for providing nodes to jobs in the batch scheduler:

1) NVDIMMs are configured 100% in App Direct and processors are in 1LM mode.

2) NVDIMM regions are configured as Device DAX by default in the Linux configuration (may be used for persistent storage as a single file).

3a) An application that cannot work without multiple files may request that the batch scheduler reconfigures regions as FSDAX.[5]

3b) An application that needs lots of volatile memory may request that the batch scheduler reconfigures a region as an additional NUMA node through System-RAM mode.[6]

This solution does not bring locality problems because each CPU still has its local DRAM explicitly available, while its local NVDIMMs may be exposed in the mode that matches the local job needs. Also, this approach is on par with current Linux kernel development towards exposing both DRAM and PMEM as explicit NUMA nodes and having ways to migrate hot pages between fast and slow memory[7].

Table 6.1 summarises the advantage of our proposal compared to 2LM memory presented in the previous Section.

[4] This is similar to what happened in many KNL clusters: some nodes were in Cache mode, others in Flat mode.

[5] Using the `ndctl` command-line tool, which does not require a reboot.

[6] Using the `daxctl` command-line tool, which does not require a reboot.

[7] https://lwn.net/Articles/787418/

Table 6.1: Advantages and drawbacks of 2LM and 1LM modes for co-scheduling jobs.

CPU Config	**2-Level-Memory**	**1-Level-Memory**
NVDIMM Config	Memory Mode ratio depends on jobs Reboot required for updating	100% App Direct
Fast/Slow Memory Management	Automatic (DRAM Cache)	Manual & Flexible (NUMA)
Storage Management	Limited to App Direct ratio	OK
Locality	May miss local memory	OK

6.4 Fine Grain Partitioning between HPC jobs

We showed in the previous section that 1LM is a good trade-off, enabling flexibility with respect to application needs and memory management. We now explain how to actually partition and expose different kinds of memory between jobs at a fine grain.

HPC resource managers may already use Linux Cgroups for partitioning CPUs between jobs [30], as well as NUMA nodes (individual nodes or amounts of memory may be dedicated to each group). This work may already be applied to partition NVDIMM-based NUMA nodes, either in 2LM or System-RAM nodes in 1LM.

However, when a single Device DAX is used, there is no way to partition it between multiple jobs. This is an issue that is addressed in the following section.

6.4.1 NVDIMMs Hardware Partitioning

As explained in Section 6.2, each NVDIMM (its App Direct part) may be exposed as an individual region or it may be interleaved with others (see Figure 6.10). Each region is exposed as a different FSDAX, Device DAX or NUMA node in Linux, which may be allocated to different jobs by the administrator. However, with only 6 channels per CPU and 1 single DCPMM per channel (128, 256 or 512GB each), there are very few possibilities for partitioning. Moreover, modifying regions requires a long reconfiguration process (minutes) and a reboot. Hence, we do not think this is a good way to partition NVDIMMs between jobs. Device DAX partitioning between different jobs actually requires synchronisation between jobs. We explain in Section 6.4.2 how resource managers may solve this issue using namespaces.

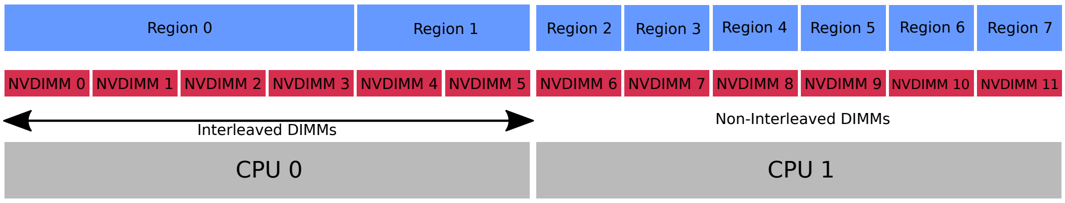


Figure 6.10: Partitioning NVDIMMs using regions and interleaving. On the first processor two interleaved regions use respectively 4 and 2 NVDIMMs. On the second processor, all NVDIMMs are exposed as individual non-interleaved regions.

6.4.2 Multidax and namespace-based software partitioning

The partitioning should rather be applied in software on top of persistent memory regions. Indeed, each region may be split into different namespaces that are configured by the administrator without requiring a reboot [101].[8] Hence, we believe that the hardware configuration should consist in one interleaved region per locality domain (CPU or SubNUMA Cluster, for good NUMA locality). The resource manager would then use namespaces for partitioning those static regions dynamically on job allocation. We observed a 1-gigabyte minimal granularity for this partitioning on Leonide platform, and we believe this is sufficient for current HPC jobs on platforms with tens of hundreds of GB of memory.

Hence, we propose to extend the strategy from Section 6.3.2:

1) NVDIMMs are configured 100% in App Direct and processors are in 1LM mode. **NVDIMM regions are interleaved at CPU level (or SubNUMA Cluster).**

2) **Jobs request one or several region namespaces from the resource manager.** Namespaces are configured as Device DAX by default in the Linux configuration (that may be used for persistent storage as a single file).

3) **Jobs specify how each namespace should be configured, having multiple namespaces in the existing regions such as shown in Figure 6.11.**

3a) An application that cannot work without multiple files may request the reconfiguration **of a namespace** as FSDAX.

[8] Using the `ndctl` command-line tool again.

3b) An application that needs lots of volatile memory may request the reconfiguration **of a namespace** as an additional NUMA node through the System-RAM mode.

If multiple namespaces from the same physical region are exposed as NUMA node, they are actually exposed as a single NUMA node[9]. Linux Cgroups may be used to partition the memory of that shared NUMA node between jobs although they still lack some features for advanced cases[10].

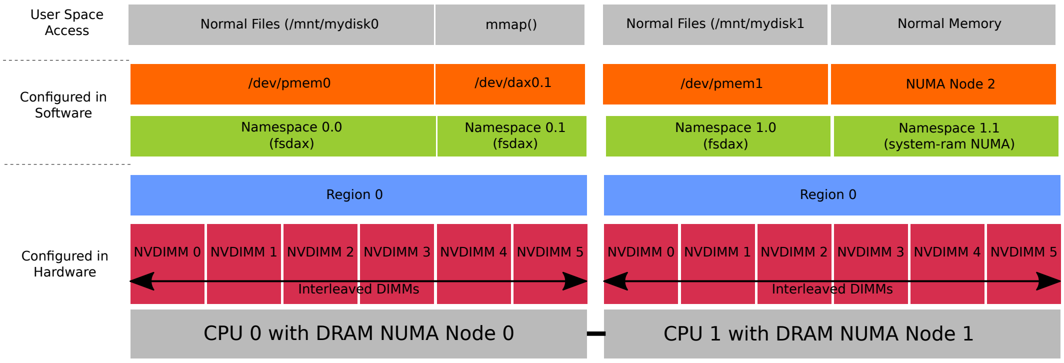


Figure 6.11: Using namespaces to partition regions between jobs requiring FSDAX, Device DAX or NUMA nodes. Each processor is configured with a single interleaved region. Software splits them between namespaces that may be configured according to jobs requirements.

Partitioning a single region into multiple DAX does not incurs in a performance penalty. Indeed, processes only map DAX pages in their virtual address spaces and access them as regular memory. The actual overhead of using multiple namespaces is their creation during job prologue (a couple of minutes).

6.4.3 Dax Locality

Finally, we look at how locality information is exposed in the proposed strategy. Indeed, even if the resource manager tries its best to allocate local namespaces to jobs, there is no guarantee that it will always be possible, and we explained in Section 2.4.2 that locality matters to performance of NVDIMMs. Hence, there is a need for the resource manager and the application to gather locality information about the different software handles that correspond to NVDIMMs.

When NVDIMMs are exposed as additional NUMA nodes, we implemented in hwloc a way to find out the corresponding local CPUs and DRAM by

[9] Each persistent memory region corresponds to a unique NUMA node in ACPI tables.

[10] Cgroups can limit a process overall memory use, and the NUMA nodes it may use, but not the memory use within a single NUMA node

looking at NUMA distances and memory target-initiator information in Linux. Figure 6.12 depicts an example of such configuration.

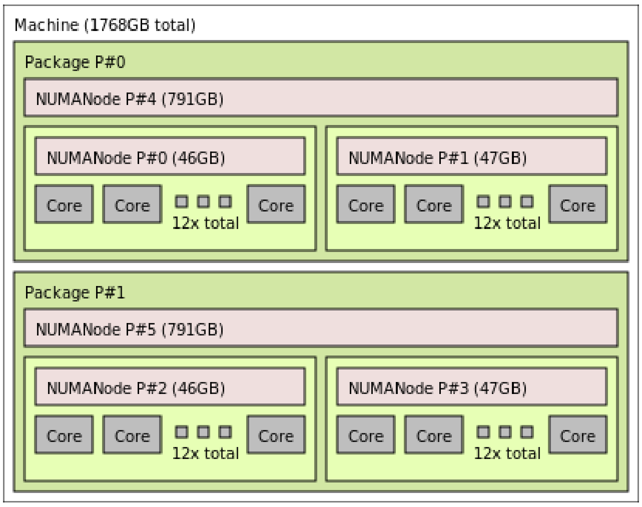


Figure 6.12: hwloc's lstopo representation of a platform with NVDIMMs exposed as additional NUMA nodes, using the System-RAM mode. Each processor has one local DRAM NUMA node per SubNUMA Cluster (e.g. #0 and #1) and a single NVDIMM NUMA node (e.g. #4). Hence, each core has two local memories.

For other cases (FSDAX, Device DAX and raw namespace), the information exposed by Linux is currently incomplete: only one local DRAM node is reported even if there are multiple of them. For instance, in Figure 6.12, they would be reported as close to NUMA node #0 only (SubNUMA Cluster) instead of both #0 and #1 (entire Package).

6.5 Discussion and Summary

HPC nodes are growing, causing co-scheduling to become necessary as soon as applications do not scale well to many cores. Indeed, it is better to fill powered-on nodes rather than powering up yet another partially-used node. However, previous work has shown than node sharing raises several performance issues, especially in the memory subsystem [93]. Many resources may be partitioned in software to avoid processes disturbing each others.

Resource managers such as SLURM are usually in charge of allocating cores and memory to jobs. They now use techniques such as Linux Cgroups for partitioning these resources between jobs [30] or containers [110]. Cache partitioning also appeared in recent processors as a way to also avoid co-existing cache pollution between applications [52]. However, it is currently not supported by DDR caches in 2LM. Fortunately, we explained that we do not believe that 2LM is a sensible choice for HPC nodes.

When NVDIMMs are used as persistent storage, the resource manager is in charge of allocating this local storage to jobs. Like any local disk in computing nodes, these FSDAX may be provisioned by the manager, for instance as explicit or automatic burst buffers [36, 106]. This local storage may also be used as a high-performance temporary storage between different jobs [36] for instance for in-situ analysis.

All these techniques are compatible with our proposal for partitioning non-volatile memory since we apply the partitioning when launching jobs (in the job prologue) and not in hardware.

We studied the different ways to use an heterogeneous memory system that contain NVDIMMS parting from KNL experiments and showing that support different use cases for different application needs requires careful hardware configuration. We have explained why 2-Level-Memory is not a convenient solution for locality-aware partitioning of NVDIMMs between jobs. We have shown why 1-Level-Memory looks like a better approach with more flexibility for memory allocation, easier configuration for the administrator and resource managers, and better locality.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

HPC has been innovating to be able to lead scientific discoveries with the aim that human beings have a better quality of life. Advances have been made in both scientific and social foundations such as self-driving model development, tracking a storm, analysing seismic waves, analysing stock trends, advancing in precision medicine, etc.

HPC has evolved over the years due to the growing need for more powerful supercomputers. In particular, the memory hierarchy of these systems has also undergone substantial changes. These changes have implied that HPC software is in need of supporting heterogeneous memory systems. That is, supercomputers may have more than one type of memory. HPC software does not have a very efficient way to expose or manage these systems, since homogeneous memory systems are normally used, so the support is still very limited.

Another unfortunate consequence of having heterogeneity on the memory system is the need for applications to be adapted so that they can properly exploit the memory system, and in this way not underutilise resources. Not all applications have the same behaviour when allocating in a determined memory of an HMS.

Currently, the technologies that make up heterogeneous memory systems (DRAM, HBM, NVDIMM, etc.) are in need of exposing their different memory attributes, since this makes them differentiated. Something that was not necessary with homogeneous memory systems. However, it is clear that the use of these memory systems by developers (even the most experienced ones) is more complex. It is necessary to adapt the existing applications and provide the necessary tools to make this transition, but it is essential to think that the development of new applications not only requires these tools. Developers now face the scenario, where their application must run on any type of heterogeneous memory system. The challenges are still many, and more systems with heterogeneous memory are being announced, so developers cannot be allowed to become a bottleneck in the face of such rapid progress.

Contributions

In this thesis we have explored axes of research related to management and data-placement in heterogeneous memory systems. Giving tools and strategies that can be used by end users. We have worked with at least two systems with different heterogeneous memory and we have done different tests with some applications.

In Chapter 3 we have designed **an extension to the hwloc interface that allows managing the complexity of memory systems** in emerging and future architectures. Due to the different kinds of memory and their different characteristics, it is a great challenge to use them in an efficient and effective way. This extension allows querying and classifying memory devices. The attributes represent characteristics that are relatively easy to understand, such as bandwidth, latency, and capacity. With this interface, we simplify building high-level abstractions to give developers more productivity and performance.

In Chapter 4 **a general strategy framework is shown to identify the affinity of application buffers towards a certain memory attribute**. Benchmarking and profiling are two methods that open the black box of memory attributes that applications can be sensitive to. This approach gives developers a method to analyse their applications either by binding them to different types of memory as well as profiling them in order to detect their affinity to a memory attribute. This approach brings an improvement in the productivity of developers since this information of sensitivity can be passed to allocators in a portable way without the need to hardwire the information of a type of memory existing in the application code.

In Chapter 5 we give to developers some **strategies to simulate heterogeneous memory systems**. We have presented techniques that allow manipulating the performance of hardware memory from the application point of view, for instance with bandwidth throttling, a pirate application, or NUMA distance latency injection. This eases the performance testing of codes on non-existing or unavailable heterogeneous memory systems. We also described ways to modify the memory subsystems exposed in the software stack, for instance by modifying ACPI tables or tweaking the hwloc topology. This allows checking the behaviour of runtime and applications in presence of different heterogeneous memory configurations.

In Chapter 6, we introduced a **Strategy for partitioning non-volatile memory between co-scheduled jobs** with possibly different needs. The emerging Intel Optane technology with its NVDIMMs that may be configured and used in may different ways raises many questions their actual use in HPC centers. Parting from KNL experiments we have showed that supporting different use cases for different application needs requires careful hardware configuration. We have explained why 2-Level-Memory is not a convenient solution for locality-aware partitioning of NVDIMMs between jobs. We have shown why 1-Level-Memory looks like a better approach with more flexibility for memory allocation, and we proposed a hardware/software strategy to make its configuration easier for the administrator and resource managers, while meeting better locality.

7.2 Future Work

Our work has allowed better understanding heterogeneous memory systems, where we were able to manipulate their configuration and also learn to operate them. We plan to develop certain areas of work already studied, while also, anticipate that this research will allow the development of solutions for the software stack that better exploit emerging hardware.

Validate and extend our work on emerging platforms. Nowadays vendors do not include reliable information of their products in HMAT, or simply they do not provide any HMAT table at all. It raises the problem of obtaining values for our memory attributes (see Chapter 3). Things are expected to improve with new platforms. More heterogeneous platforms are announced. Intel will have heterogeneous systems that have DRAM and HBM soon, ARM is also venturing into hybrids of DRAM and HBM, and POWER with hybrids of DRAM and NVDIMMs. Once these tables are actually available in more platforms, we will be able to validate more precisely the usefulness of our approach and of our attributes, and possibly add new attributes describing new characteristics of memory. Through refining the work done on this thesis, we should simplify the developer's work when looking for which memory is better for what.

Extend our allocation policies to handle more application requirements, keeping the focus on the management of heterogeneous memory systems. The main reason is that the memory wall is going to continue, and hybrid memory technologies are going to become mainstream. In Chapter 3 the work has consisted of exposing heterogeneous memory systems to applications providing low-level software support. But this support with the applications and runtime system may still be extended. For instance, we envision the need to migrate buffers between kinds of memory, for instance when the application access pattern varies between steps. Since migration is known to be expensive, we will need to find a trade-off between what the performance gain from migrating and what we lose because of the migration overhead.

Make sensitivity a quantitative metric. As shown in Chapter 4, we have used simple understanding memory metrics that give us context about the sensitivity of an application about Latency, Bandwidth and Capacity. However, the simplicity of these metrics could be not enough in some cases to detect an affinity. For instance, bandwidth throttling presented in Chapter 5, could become a starting point to develop a more complex metric that improves the selection of appropriate memory targets by quantifying bandwidth sensitivity on an application instead of just determining if it is sensitive or not.

Define high-level metrics. Low-level metrics such as Latency, Bandwidth and Capacity are useful for runtime developers that are aware of the hardware behavior as presented in Chapter 3. However, this may not be appropriate for applications developers, who are usually not hardware experts. Hence, some high-level metrics might be needed, for instance to characterize memory access patterns with strides, dimensions, etc.

Intelligent runtime systems. There is a long path to allow a runtime to employ various strategies to place data depending on developer input. The team already work on this topic with OpenMP developers from RWTH Aachen University in the H2M project. This thesis work serves as a starting point and will be leveraged into high-level runtimes/applications. First OpenMP allocators, that may also request high-bandwidth, low latency or high-capacity target memory, will be mapped on top of the heterogeneous allocator presented in Chapter 4. Then we will develop runtime heuristics to exploit heterogeneous memory systems for dynamic abstract data structures. It will be based on a memory performance model supporting different kinds of memory to enable runtimes to employ heuristics.

Static code analysis for taking allocation decisions. In Chapter 4, we showed two strategies to identify the sensitivities of the application buffers and thus have a criterion from which the allocator can feed. The first one has consisted of binding the applications to different types of memory and comparing their performance to then have an allocation criteria. The second one consisted of profiling the application, where we can identify the main memory objects and their sensitivity to predefined metrics such as bandwidth, latency and capacity which not requires too much executions to determine the allocation criteria. We consider there are two more possible way to be explored. Using static code analysis, which consists in studying the source code, for instance during compilation, is another way to discover sensitivity. This step may provide the compiler and/or runtime with additional information about an application. For instance, what is going to happen in the future with a data buffer? In our approach, the plan would rather have the compiler insert annotations in the code to tell the runtime where to allocate each buffer. This requires significant work, with collaboration between compilers and runtimes, but recent compiler advances such as plugins in LLVM make prototyping more accessible.

In the same fashion as the previous paragraph, we consider **Artificial Intelligence (AI) for taking allocation decisions**. It has successfully been used for deciding of NUMA memory placement [21] and we believe it may also be applied to memory placement in HMS. Characterising the interaction of applications with different kinds of memories and applying learning techniques may allow to identify common behaviours and predict a good allocation criteria. However, this raises the question of having enough data to feed the AI model, which means many different heterogeneous platforms and/or many different applications.

Appendix

Appendix A

Platform Characteristics

For the experiments carried out in this thesis, we use computation nodes from the Experimental Platform Dalton and the Federative Platform for Research in Computer Science and Mathematics(PlaFRIM) [82].

A.1 Kona: Xeon Phi Knights Landing (KNL)

There are four Kona systems that have the particularity of having high-bandwidth memory (16GB MCDRAM) on the processor chip and DRAM memory. The 64 cores of this processor (Intel Knights Landing Xeon Phi 7230), clocked at 1.3GHz, are grouped in tiles of 2 cores sharing an L2 cache. These tiles are interconnected through a two-dimensional cartesian mesh.

The servers are configurable at start-up to expose as a single NUMA domain, or 2 or 4 Sub-NUMA-Clusters (SNC), each composed of MCDRAM memory and DRAM memory.

Some Kona nodes are configured to delegate the hardware the management of the MCDRAM as a last level of cache (Cache mode) when defining the memory mode in the BIOS, but these nodes have not been used in this thesis. Only nodes exposing the MCDRAM as a separate NUMA node (Flat mode) were used.

A.1.1 Kona01

The topology of Kona01 machine, configured with a single NUMA domain, is provided in Figure A.2 and the system configuration is shown in Table A.1.

Processor	Xeon Phi 7230
Figure	A.2
Frequency (GHz)	1.3
Number of Core	64
Core interconnection topology	2D mesh
Number of *Socket*	1
Number of NUMA domains	1
Cluster mode	Quadrant
Memory mode	Flat
Hyper-Threading	Yes
Turbo Boost	Yes

Figure A.1: Kona01 system characteristics and configuration.

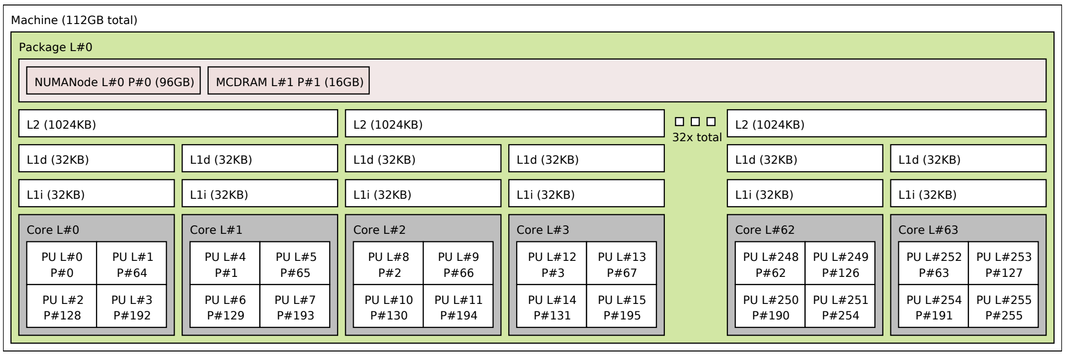


Figure A.2: Topology of the Kona01 node.

A.1.2 Kona03

The topology of Kona03 machine divided into 4 NUMA domains is provided in Figure A.4 and the system configuration is shown in Table A.3. Only the cores of the first SubNUMA Cluster are shown for simplicity.

Processor	Xeon Phi 7230
Figure	A.4
Frequency (GHz)	1.3
Number of Cores	64
Core interconnection topology	2D mesh
Number of *Sockets*	1
Number of NUMA domains	4
Sockets interconnection technology	N/A
Cluster mode	SNC4
Memory mode	Flat
Hyper-Threading	Yes
Turbo Boost	Yes

Figure A.3: Kona03 system characteristics and configuration.

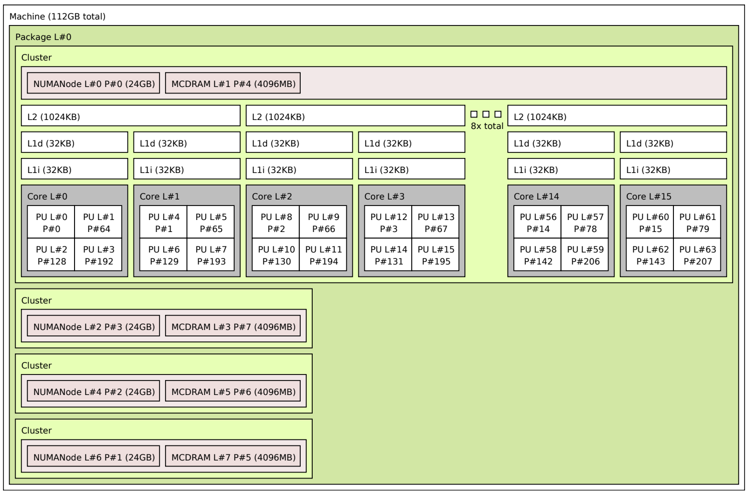


Figure A.4: Topology of the Kona03 node. Only cores of the first SubNUMA Cluster are shown.

A.2 Leonide: dual Intel Xeon Gold 6230 with NVDIMMs

The system has the particularity of supporting high-capacity memory (NVDIMMs) and DRAM memory. It has 2 physical processors (Intel Xeon Cascade Lake 6230), 20 cores per processor, clocked at 2.1GHz. The machine as shown in Table A.5 is configurable at start-up to expose one NUMA domain or 2 Sub-NUMA-Cluster (SNC) per CPU. DRAM is split accordingly, while NVDIMMs are always shared by the entire CPU. Processors can also be configured to work in 2-Level-Memory mode or 1-Level-Memory mode. NVDIMMs are Intel *Data Center Persistent Memory Modules*, 6×128GB per CPU. They can be configured to use Memory mode, App Direct mode or System-RAM mode. Different mode's combinations are shown in Figures A.6, A.7, A.8 and A.9.

Processor	Xeon Gold 6230
Frequency (GHz)	2.1
Number of Cores	40
Number of *Sockets*	2
Number of NUMA domains	2 or 4 DDR, up to 2 NVDIMM
Sockets interconnection technology	N/A
SNC mode	SNC2, no SNC
Processor mode	1LM, 2LM
Memory mode	Memory, App Direct, System-RAM
Hyper-Threading	Yes

Figure A.5: Leonide system characteristics and configuration.

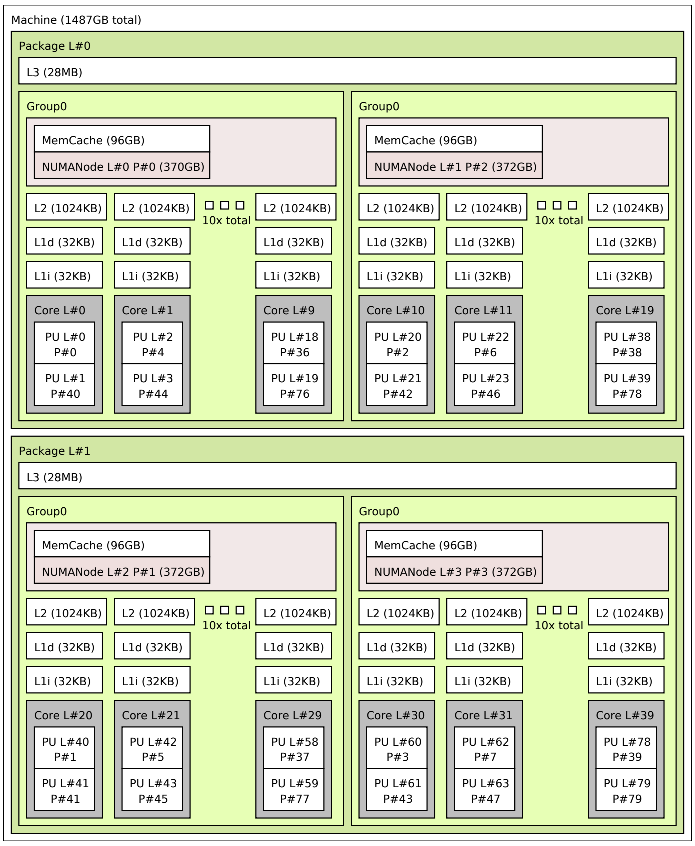


Figure A.6: Intel Xeon Cascade Lake in SNC-2 mode, in 2-Level-Memory mode, using NVDIMMs 100% in Memory mode, and using DRAM as cache in front of NVDIMMs.

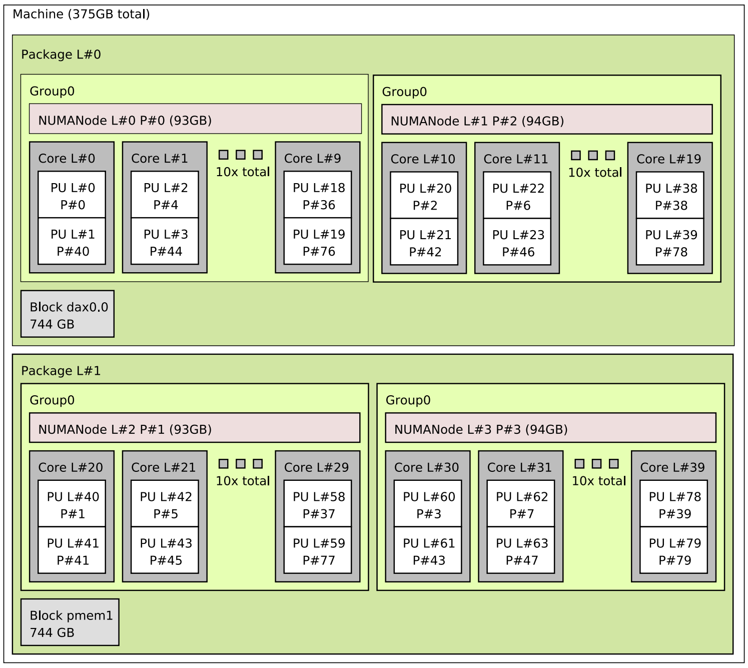


Figure A.7: Intel Xeon Cascade Lake in SNC-2 mode, in 1-Level-Memory mode, using NVDIMMs 100% in App Direct mode, NVDIMMs in Package #0 are setup in Device DAX (devdax device `dax0.0`), and NVDIMMs in Package #1 are setup in File System DAX (fsdax device `pmem1`).

A.3 Souris: SGI Altix UV 2000

This SGI Altix UV 2000 system has the particularity of have 12 sockets nodes, 8 cores per processor, clocked at 2.6GHz. It has a total of 3TB of memory (16GB/core) as shown in Table A.10. Processors are grouped by pair on physical *Blades* that correspond to *Groups* in Figure A.11. Hence, there is at least 3 levels of NUMA locality: local memory, memory in the neighbour CPU of the same blade, and memory in other blades. However, the interconnect between blades is not a complete graph, hence some blades are 2-hops away.

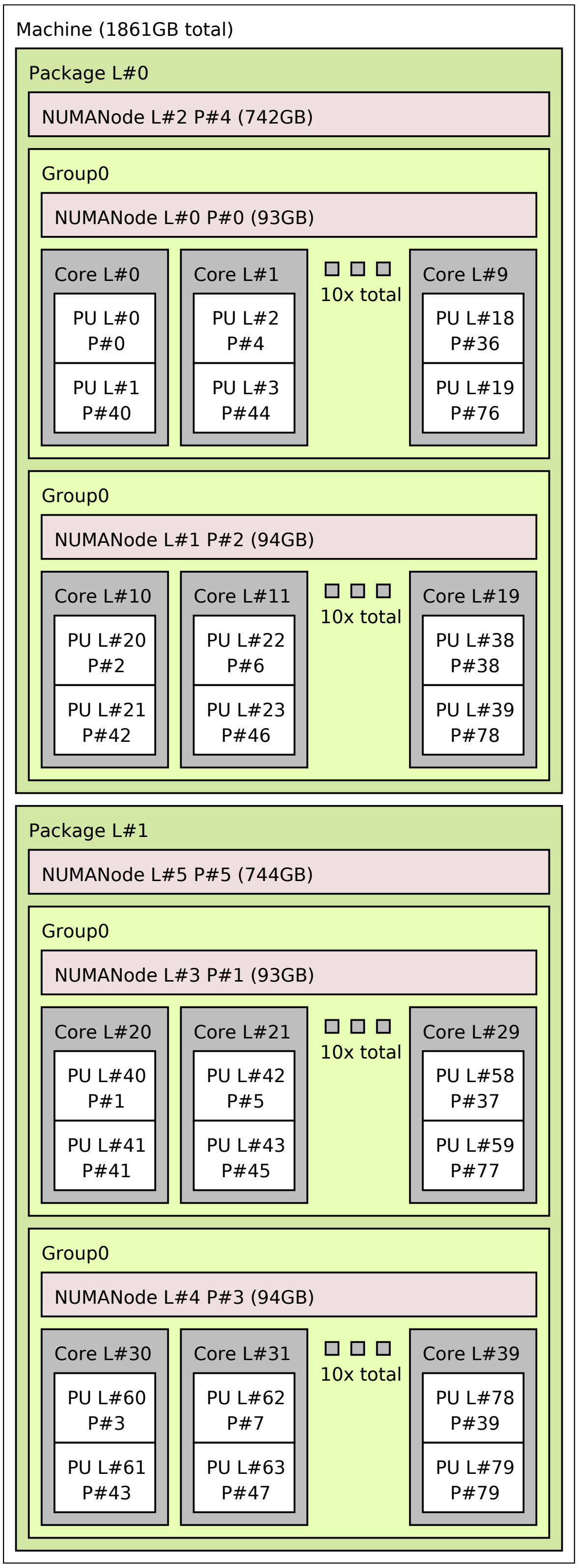


Figure A.8: Intel Xeon Cascade Lake in SNC-2 mode, in 1-Level-Memory mode, using NVDIMMs in System-RAM mode (NUMA nodes P#4 and P#5).

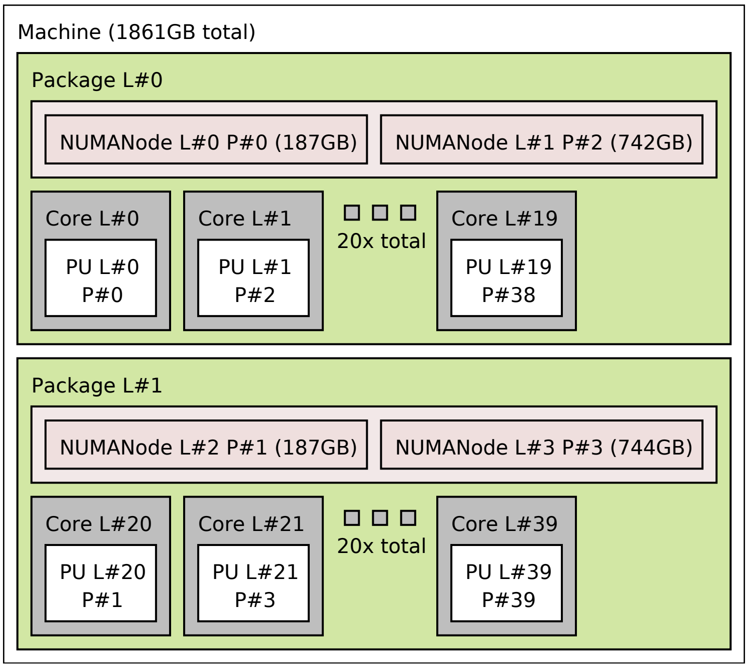


Figure A.9: Intel Xeon Cascade Lake not in SNC mode, in 1-Level-Memory mode, using NVDIMMs in System-RAM mode (NUMA nodes P#2 and P#3).

Processor	Ivy-Bridge Xeon E5-4620v2
Figure	A.11
Frequency (GHz)	2.6
Number of Cores	96
Number of *Sockets*	12
Number of NUMA domains	12
Hyper-Threading	Yes
Turbo Boost	Yes

Figure A.10: Souris system characteristics and configuration.

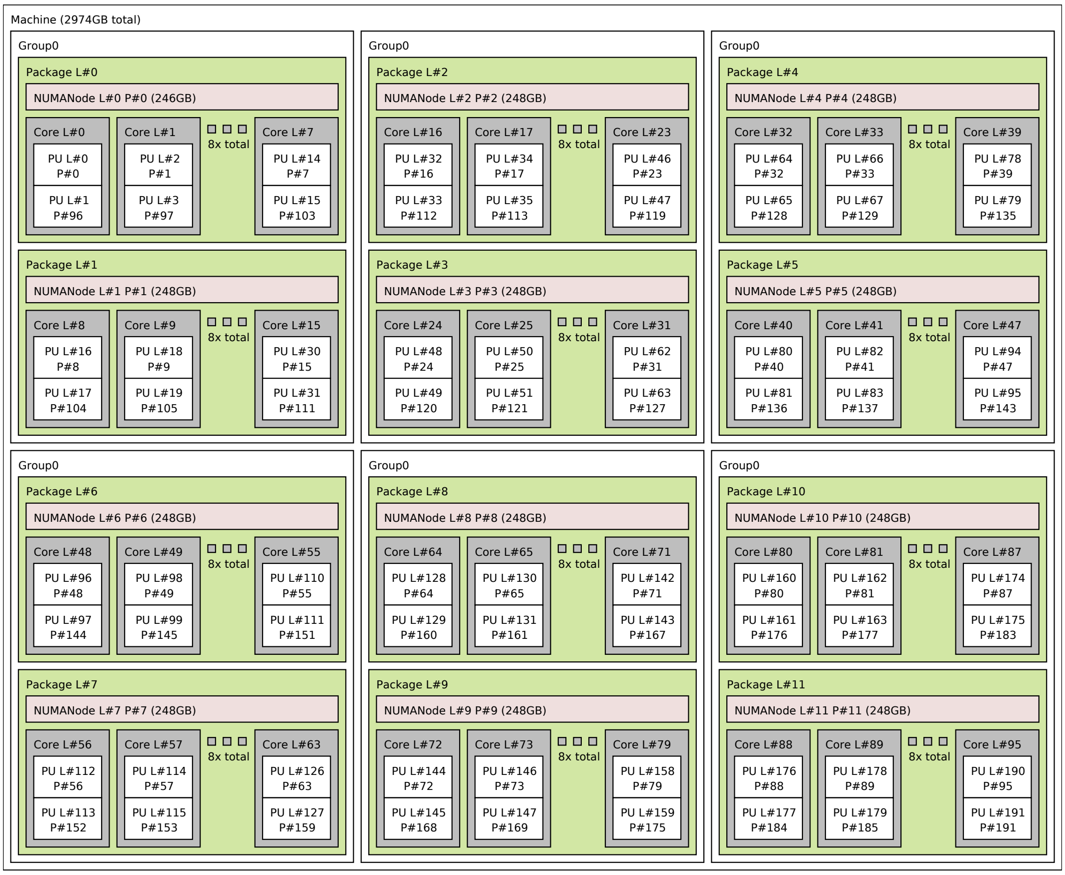


Figure A.11: Topology of the Souris node.

Bibliography

[1] Giovanni Agosta, William Fornaciari, Giuseppe Massari, Anna Pupykina, Federico Reghenzani, and Michele Zanella. Managing heterogeneous resources in HPC systems. In *ACM International Conference Proceeding Series*, pages 7–12, New York, New York, USA, 2018. ACM Press.

[2] Mulya Agung, Muhammad Alfian Amrizal, Ryusuke Egawa, and Hiroyuki Takizawa. DeLoc: A Locality and Memory-Congestion-Aware Task Mapping Method for Modern NUMA Systems. *IEEE Access*, 8:6937–6953, 2020.

[3] William Allcock, Bennett Bernardoni, Colleen Bertoni, Neil Getty, Joseph Insley, Michael E. Papka, Silvio Rizzi, and Brian Toonen. RAM as a network managed resource. In *Proceedings - 2018 IEEE 32nd International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2018*, pages 99–106, 2018.

[4] George Almási, Ralph Bellofatto, José Brunheroto, Călin Caşcaval, José G. Castanos, Luis Ceze, Paul Crumley, C. Christopher Erway, Joseph Gagliano, Derek Lieber, Xavier Martorell, José E. Moreira, Aida Sanomiya, and Karin Strauss. An overview of the blue gene/L system software organization. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2790:543–555, 2004.

[5] Mohammed Almiyad and Anas Alhasan. An automation engine to improve seismic operations in exploration. *International Petroleum Technology Conference 2020, IPTC 2020*, jan 2020.

[6] Lluc Alvarez, Eduard Ayguade, Marc Casas, Mateo Valero, Jesus Labarta, and Miquel Moreto. Runtime-guided management of stacked DRAM memories in task parallel programs. In *Proceedings of the International Conference on Supercomputing*, pages 218–228. Association for Computing Machinery, jun 2018.

[7] AMD. High Bandwidth Memory, 2016. `http://www.amd.com/en-us/innovations/software-technologies/hbm`.

[8] Amd. AMD64 Technology Platform Quality of Service Extensions. Technical report, 2018.

[9] Mohamed Arafa, Bahaa Fahim, Sailesh Kottapalli, Akhilesh Kumar, Lily P. Looi, Sreenivas Mandava, Andy Rudoff, Ian M. Steiner, Bob Valentine, Geetha Vedaraman, and Sujal Vora. Cascade Lake: Next Generation Intel Xeon Scalable Processor. *IEEE Micro*, 39(2):29–36, 2019.

[10] Luna Backes and Daniel A. Jiménez. The impact of cache inclusion policies on cache management techniques. *ACM International Conference Proceeding Series*, pages 428–438, 2019.

[11] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose Carlos Sancho. A performance evaluation of the Nehalem quad-core processor for scientific computing. In *Parallel Processing Letters*, volume 18, pages 453–469. World Scientific Publishing Company, dec 2008.

[12] Taylor Barnes, Brandon Cook, Jack Deslippe, Douglas Doerfler, Brian Friesen, Yun He, Thorsten Kurth, Tuomas Koskela, Mathieu Lobet, Tareq Malas, Leonid Oliker, Andrey Ovsyannikov, Abhinav Sarje, Jean Luc Vay, Henri Vincenti, Samuel Williams, Pierre Carrier, Nathan Wichmann, Marcus Wagner, Paul Kent, Christopher Kerr, and John Dennis. Evaluating and optimizing the NERSC workload on knights landing. In *Proceedings of PMBS 2016: 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems - Held in conjunction with SC 2016: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 43–53, 2017.

[13] D A Beckingsale and R D Hornung. Umpire : Status Report and Future Development Plan. 2018.

[14] D. A. Beckingsale, M. J. McFadden, J. P.S. Dahm, R. Pankajakshan, and R. D. Hornung. Umpire: Application-focused management and coordination of complex hierarchical memory. *IBM Journal of Research and Development*, 64(3-4), may 2020.

[15] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing, PDP 2010*, pages 180–186, 2010.

[16] Christopher Cantalupo, Jeff R Hammond, and Simon Hammond. User Extensible Heap Manager for Heterogeneous Memory Platforms and Mixed Memory Policies. (2):1–17, 2015.

[17] Suma George Cardwell, Craig Vineyard, Willam Severa, Frances S. Chance, Frederick Rothganger, Felix Wang, Srideep Musuvathy, Corinne Teeter, and James B. Aimone. Truly heterogeneous HPC: Co-design to achieve what science needs from HPC. In *Communications in Computer and Information Science*, volume 1315 CCIS, pages 349–365. Springer, Cham, aug 2021.

[18] Gopinath Chennupati, Nandakishore Santhi, Stephan Eidenbenz, and Sunil Thulasidasan. An analytical memory hierarchy model for performance prediction. In *Proceedings - Winter Simulation Conference*, pages 908–919. Institute of Electrical and Electronics Engineers Inc., jun 2017.

[19] Intel Corporation. Introduction to Intel ® Optane ™ DC Technology Frequently Asked Questions (FAQ). Technical report, 2019.

[20] Nicolas Denoyelle, Brice Goglin, Aleksandar Ilic, Emmanuel Jeannot, and Leonel Sousa. Modeling non-uniform memory access on large compute nodes with the cache-aware roofline model. *IEEE Transactions on Parallel and Distributed Systems*, 30(6):1374–1389, 2019.

[21] Nicolas Denoyelle, Brice Goglin, Emmanuel Jeannot, and Thomas Ropars. Data and thread placement in NUMA architectures: A statistical learning approach. *ACM International Conference Proceeding Series*, aug 2019.

[22] Nicolas Denoyelle, John Tramm, Kazutomo Yoshii, Swann Perarnau, and Pete Beckman. Numa-Aware Data Management for Neutron Cross Section Data in Continuous Energy Monte Carlo Neutron Transport Simulation. *EPJ Web of Conferences*, 247:04020, 2021.

[23] Ulrich Detert and Gerd Hofemann. CRAY X-MP and Y-MP memory performance. *Parallel Computing*, 17(4-5):579–590, jul 1991.

[24] Zhuohui Duan, Haikun Liu, Xiaofei Liao, Hai Jin, Wenbin Jiang, and Yu Zhang. HiNUMA: NUMA-aware data placement and migration in hybrid memory systems. In *Proceedings - 2019 IEEE International Conference on Computer Design, ICCD 2019*, pages 367–375. Institute of Electrical and Electronics Engineers Inc., nov 2019.

[25] Loïc Duflot, Olivier Levillain, and Benjamin Morin. Acpi: Design principles and concerns. Technical report, 2009.

[26] Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *Journal of the ACM*, 44(6):779–805, nov 1997.

[27] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. Cache pirating: Measuring the curse of the shared cache. In *Proceedings of the International Conference on Parallel Processing*, pages 165–175, 2011.

[28] Fujitsu. Toca do Tux: Fujitsu trabalha em seu novo supercomputador. `http://www.tocadotux.com.br/2019/12/fujitsu-trabalha-em-seu-novo.html`.

[29] Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. Challenges of memory management on modern NUMA systems. *Communications of the ACM*, 58(12):59–66, dec 2015.

[30] Yiannis Georgiou and Matthieu Hautreux. SLURM Resources isolation through cgroups (2011). *Georgiou, Y., Hautreux*, 2011.

[31] Richard J. Glassock. A Personal Perspective on the State of the Art in Research in Glomerulonephritis. In Peter H Welch, Frederick R M Barnes, Jan F Broenink, Kevin Chalmers, Jan Bækgaard Pedersen, and Adam T Sampson, editors, *Chronic Renal Disease*, pages 5–7. Open Channel Publishing Ltd., 1985.

[32] M. D. Godfrey and D. F. Hendry. The Computer as von Neumann Planned It. *IEEE Annals of the History of Computing*, 15(1):11–21, 1993.

[33] Brice Goglin. Exposing the locality of heterogeneous memory architectures to HPC applications. In *ACM International Conference Proceeding Series*, volume 03-06-October-2016, pages 30–39, 2016.

[34] Jimmy Handy and Tom Coughlin. The Future of Low Latency Why Near Memory Requires a New Interface [White Paper]. *SNIA Persistent Memory + Computational Storage Summit*, 2021.

[35] Dave Hansen. Allow persistent memory to be used like normal RAM. `https://lwn.net/Articles/776921/`.

[36] Dave Henseler, Benjamin Landsteiner, Doug Petesch, Cornell Wright, and Nicholas J. Wright. Architecture and Design of Cray DataWarp. *CUG '16 Proceedings of the Cray User Group*, 2016.

[37] HPCwire. It's Fugaku vs. COVID-19: How the World's Top Supercomputer Is Shaping Our New Normal. https://www.hpcwire.com/2020/11/09/its-fugaku-vs-covid-19-how-the-worlds-top-supercomputer-is-shaping-our-new-normal/.

[38] W C Hsu. Register allocation and code scheduling for load/store architectures.

[39] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, vol. 1-3. (253665, 325383, 325384):366–369, 2016.

[40] Intel. Intel® Xeon Phi™ Processor x200 Product Family Datasheet, 2017. https://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-processor-x200-product-family-datasheet.html.

[41] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv*, pages 1–36, 2019.

[42] Jikku Jeemon. Pipelined 8-bit RISC Processor Design using Verilog HDL on FPGA. *2016 IEEE International Conference on Recent Trends in Electronics, Information and Communication Technology, RTEICT 2016 - Proceedings*, 4(6):2023–2027, 2017.

[43] Yichen Jia and Feng Chen. From Flash to 3D XPoint: Performance Bottlenecks and Potentials in RocksDB with Storage Evolution. In *Proceedings - 2020 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2020*, pages 192–201. Institute of Electrical and Electronics Engineers Inc., aug 2020.

[44] Kernel.org. 20. User Interface for Resource Control feature — The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/x86/resctrl.html.

[45] Dounia Khaldi and Barbara Chapman. Towards automatic HBM allocation using LLVM: A case study with knights landing. In *Proceedings of LLVM-HPC 2016: The 3rd Workshop on the LLVM Compiler Infrastructure in HPC - Held in conjunction with SC 2016: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 12–20. Institute of Electrical and Electronics Engineers Inc., feb 2017.

[46] David B Kirk and Wen-mei W Hwu. Programming Massively Parallel Processors. *Programming Massively Parallel Processors*, 2013.

[47] Christoph Lameter. NUMA (Non-Uniform Memory Access): An overview: NUMA becomes more common because memory controllers get close to execution units on microprocessors. *Queue*, 11(7):40–51, jul 2013.

[48] Michael Kenneth Lang. Simplified Interface to Complex Memory (SICM) FY19 Project Review. Technical report, Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), oct 2019.

[49] Donghyuk Lee, Saugata Ghose, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Simultaneous multi-layer access: Improving 3d-stacked memory bandwidth at low cost. *ACM Transactions on Architecture and Code Optimization*, 12(4):1–29, jan 2016.

[50] Ang Li, Weifeng Liu, Mads R.B. Kristensen, Brian Vinter, Hao Wang, Kaixi Hou, Andres Marquez, and Shuaiwen Leon Song. Exploring and analyzing the real impact of modern on-package memory on hpc scientific kernels. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017*, (iii), 2017.

[51] Fujitsu Limited. Supercomputer " Fugaku ". *Top500*, 1(1):1, 2020.

[52] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. *Proceedings - International Symposium on High-Performance Computer Architecture*, pages 367–378, 2008.

[53] Haikun Liu, Renshan Liu, Xiaofei Liao, Hai Jin, Bingsheng He, and Yu Zhang. Object-Level Memory Allocation and Migration in Hybrid Memory Systems. *IEEE Transactions on Computers*, 69(9):1401–1413, sep 2020.

[54] Jihang Liu and Shimin Chen. Initial experience with 3D XPoint main memory. *Distributed and Parallel Databases*, 38(4):865–880, 2020.

[55] LLNL. LLNL_BGL_Diagram.png (600×409). `https://upload.wikimedia.org/wikipedia/commons/b/b4/LLNL_BGL_Diagram.png`.

[56] Lily P. Looi. (5) Intel Optane DC Persistent Memory Performance Overview - YouTube. `https://www.youtube.com/watch?v=UTVt_AZmWjM`.

[57] O. G. Lorenzo, T. F. Pena, J. C. Cabaleiro, J. C. Pichel, and F. F. Rivera. 3DyRM: a dynamic roofline model including memory latency information. *Journal of Supercomputing*, 70(2):696–708, nov 2014.

[58] Lwn. Top-tier memory management [LWN.net]. `https://lwn.net/Articles/857133/`.

[59] Norman Macrae and Herbert F. York. John Von Neumann: The Scientific Genius Who Pioneered the Modern Computer, Game Theory, Nuclear Deterrance, and Much More . *Physics Today*, 46(10):119–120, oct 1993.

[60] John D McCalpin. Memory bandwidth: Stream benchmark performance results, 2002. `http://www.cs.virginia.edu/stream/`.

[61] John David Mccalpin and John D Mccalpin. Memory bandwidth and machine balance in high performance computers Technology Trends in High Performance Computing View project Dynamics of Mesoscale Ocean Circulation View project Memory Bandwidth and Machine Balance in Current High Performance Computers. Technical report.

[62] Neil A. Mehta, Rahulkumar Gayatri, Yasaman Ghadar, Christopher Knight, and Jack Deslippe. Evaluating Performance Portability of OpenMP for SNAP on NVIDIA, Intel, and AMD GPUs Using the Roofline Methodology. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 12655 LNCS, pages 3–24. Springer, Cham, nov 2021.

[63] Daniel Molka, Robert Schöne, Daniel Hackenberg, and Wolfgang E. Nagel. Detecting memory-boundedness with hardware performance counters. *ICPE 2017 - Proceedings of the 2017 ACM/SPEC International Conference on Performance Engineering*, 17:27–38, 2017.

[64] Stéphanie Moreaud and Brice Goglin. Impact of NUMA effects on high-speed networking with multi-OPTERON machines. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 24–29. ACTA Press, nov 2007.

[65] Richard Murphy. On the effects of memory latency and bandwidth on supercomputer application performance. In *Proceedings of the 2007 IEEE International Symposium on Workload Characterization, IISWC*, pages 35–43, 2007.

[66] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the Graph 500 Introduction : Why Another. *Cray Users Gr.*, 19:45–74, 2010.

[67] Ravi Nair. Evolution of Memory Architecture. *Proceedings of the IEEE*, 103(8):1331–1345, aug 2015.

[68] Aditya Narayan, Tiansheng Zhang, Shaizeen Aga, Satish Narayanasamy, and Ayse Coskun. MOCA: Memory object classification and allocation in heterogeneous memory systems. *Proceedings - 2018 IEEE 32nd International Parallel and Distributed Processing Symposium, IPDPS 2018*, pages 326–335, 2018.

[69] Anant Vithal Nori, Jayesh Gaur, Siddharth Rai, Sreenivas Subramoney, and Hong Wang. Criticality aware tiered cache hierarchy: A fundamental relook at multi-level cache hierarchies. In *Proceedings - International Symposium on Computer Architecture*, pages 96–109. Institute of Electrical and Electronics Engineers Inc., jul 2018.

[70] A. Nowak. Opportunities and choice in a new vector era. *Journal of Physics: Conference Series*, 523(1):012002, jun 2014.

[71] Wilfried Oed. Cray Y-MP C90: System features and early benchmark results. *Parallel Computing*, 18(8):947–954, aug 1992.

[72] Ismail Oukid and Lucas Lersch. On the Diversity of Memory and Storage Technologies. *Datenbank-Spektrum*, 18(2):121–127, 2018.

[73] Jinsu Park, Seongbeom Park, and Woongki Baek. CoPart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers. In *Proceedings of the 14th EuroSys Conference 2019*, volume 19, New York, NY, USA, 2019. ACM.

[74] Onkar Patil, Latchesar Ionkov, Jason Lee, Frank Mueller, and Michael Lang. Performance characterization of a DRAM-NVM hybrid memory architecture for HPC applications using intel optane DC persistent memory modules. *ACM International Conference Proceeding Series*, pages 288–303, 2019.

[75] Ashutosh Pattnaik, Xulong Tang, Onur Kayiran, Adwait Jog, Asit Mishra, Mahmut T. Kandemir, Anand Sivasubramaniam, and Chita R. Das. Opportunistic computing in GPU architectures. In *Proceedings - International Symposium on Computer Architecture*, pages 210–223, 2019.

[76] Antonio J. Pena and Pavan Balaji. Toward the efficient use of multiple explicitly managed memory subsystems. In *2014 IEEE International Conference on Cluster Computing, CLUSTER 2014*, pages 123–131. Institute of Electrical and Electronics Engineers Inc., nov 2014.

[77] Ivy Bo Peng, Roberto Gioiosa, Gokcen Kestor, Pietro Cicotti, Erwin Laure, and Stefano Markidis. Exploring the performance benefit of hybrid memory system on HPC environments. In *Proceedings - 2017 IEEE*

31st International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2017, pages 683–692. Institute of Electrical and Electronics Engineers Inc., jun 2017.

[78] Ivy Bo Peng, Roberto Gioiosa, Gokcen Kestor, Jeffrey S. Vetter, Pietro Cicotti, Erwin Laure, and Stefano Markidis. Characterizing the performance benefit of hybrid memory system for HPC applications. *Parallel Computing*, 76:57–69, aug 2018.

[79] Swann Perarnau, Brice Videau, Nicolas Denoyelle, Florence Monna, Kamil Iskra, and Pete Beckman. Explicit data layout management for autotuning exploration on complex memory topologies. In *Proceedings of MCHPC 2019: Workshop on Memory Centric High Performance Computing - Held in conjunction with SC 2019: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 58–63. Institute of Electrical and Electronics Engineers Inc., nov 2019.

[80] Aleksey Pesterev, Nickolai Zeldovich, and Robert T. Morris. Locating cache performance bottlenecks using data profiling. In *EuroSys'10 - Proceedings of the EuroSys 2010 Conference*, pages 335–348, New York, New York, USA, 2010. ACM Press.

[81] Simon Pickartz, Jens Breitbart, and Stefan Lankes. Co-scheduling on Upcoming Many-Core Architectures. *Cosh 2017*, page 2017, 2017.

[82] PlaFRIM. PlaFRIM – Plateforme Fédérative pour la Recherche en Informatique et Mathématiques. `https://www.plafrim.fr/`.

[83] Constantin Pohl and Kai Uwe Sattler. Joins in a heterogeneous memory hierarchy: Exploiting high-bandwidth memory. In *14th International Workshop on Data Management on New Hardware, DaMoN 2018*, pages 1–10, New York, NY, USA, jun 2018. Association for Computing Machinery, Inc.

[84] Savíns Puertas-Martín, Antonio J. Banegas-Luna, María Paredes-Ramos, Juana L. Redondo, Pilar M. Ortigosa, Ol'ha O. Brovarets', and Horacio Pérez-Sánchez. Is high performance computing a requirement for novel drug discovery and how will this impact academic efforts? *Expert Opinion on Drug Discovery*, 15(9):981–986, sep 2020.

[85] Richard M. Russell. The CRAY-1 Computer System. *Communications of the ACM*, 21(1):63–72, jan 1978.

[86] Solmaz Salehian and Yonghong Yan. Evaluation of knight landing high bandwidth memory for HPC workloads. In *Proceedings of IA3 2017: 7th*

Workshop on Irregular Applications: Architectures and Algorithms, Held in conjunction with SC 2017: The International Conference for High Performance Computing, Networking, Storage and Analysis. Association for Computing Machinery, Inc, nov 2017.

[87] Rommel Sánchez Verdejo, Kazi Asifuzzaman, Milan Radulovic, Petar Radojković, Eduard Ayguadé, and Bruce Jacob. Main memory latency simulation: The missing link. In *ACM International Conference Proceeding Series*, page 10, New York, NY, USA, 2018. ACM.

[88] Mitsuhisa Sato, Yutaka Ishikawa, Hirofumi Tomita, Yuetsu Kodama, Tetsuya Odajima, Miwako Tsuji, Hisashi Yashiro, Masaki Aoki, Naoyuki Shida, Ikuo Miyoshi, Kouichi Hirai, Atsushi Furuya, Akira Asato, Kuniki Morita, and Toshiyuki Shimizu. Co-design for a64fx manycore processor and 'Fugaku'. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, volume 2020-November. IEEE Computer Society, nov 2020.

[89] Steve Scargall. Volatile Use of Persistent Memory. In *Programming Persistent Memory*, pages 155–186. Apress, 2020.

[90] Ada Sedova, John D. Eblen, Reuben Budiardja, Arnold Tharrington, and Jeremy C. Smith. High-performance molecular dynamics simulation for biological and materials sciences: Challenges of performance portability. *Proceedings of P3HPC 2018: International Workshop on Performance, Portability and Productivity in HPC, Held in conjunction with SC 2018: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, feb 2019.

[91] Hwajeong Seo, Kyuhwang An, and Hyeokdong Kwon. Compact LEA and HIGHT implementations on 8-bit AVR and 16-bit MSP processors. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 11402 LNCS, pages 253–265. Springer Verlag, aug 2019.

[92] Harald Servat, Antonio J. Pena, German Llort, Estanislao Mercadal, Hans Christian Hoppe, and Jesus Labarta. Automating the Application Data Placement in Hybrid Memory Systems. *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, 2017-September:126–136, sep 2017.

[93] Nikolay A. Simakov, Robert L. DeLeon, Joseph P. White, Thomas R. Furlani, Martins Innus, Steven M. Gallo, Matthew D. Jones, Abani Patra, Benjamin D. Plessinger, Jeanette Sperhac, Thomas Yearke, Ryan Rathsam, and Jeffrey T. Palmer. A quantitative analysis of node sharing

on HPC clusters using XDMoD application kernels. In *ACM International Conference Proceeding Series*, volume 17-21-July-2016, pages 1–8, New York, NY, USA, jul 2016. Association for Computing Machinery.

[94] Margaret L. Simmons and Harvey J. Wasserman. Performance comparison of the CRAY-2 and CRAY X-MP/416 supercomputers. *The Journal of Supercomputing*, 4(2):153–167, jun 1990.

[95] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen Chen Liu. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro*, 36(2):34–46, 2016.

[96] Nitish Kumar Srivastava and Akshay Dilip Navalakha. Pointer-Chase Prefetcher for Linked Data Structures. *CoRR*, abs/1801.0, jan 2018.

[97] Iulia Stirb. Improving runtime performance and energy consumption through balanced data locality with NUMA-BTLP and NUMA-BTDM static algorithms for thread classification and thread type-aware mapping. *International Journal of Computational Science and Engineering*, 22(2-3):200–210, 2020.

[98] Dmitry Suplatov, Maxim Shegay, Yana Sharapova, Ivan Timokhin, Nina Popova, Vladimir Voevodin, and Vytas Švedas. Co-designing HPC-systems by computing capabilities and management flexibility to accommodate bioinformatic workflows at different complexity levels. *Journal of Supercomputing*, pages 1–17, apr 2021.

[99] Sam Ainsworth Timothy and M. Jones. Software prefetching for indirect memory accesses. In *CGO 2017 - Proceedings of the 2017 International Symposium on Code Generation and Optimization*, pages 305–317. Institute of Electrical and Electronics Engineers Inc., feb 2017.

[100] Jan Treibig, Georg Hager, and Gerhard Wellein. LIKWID: Lightweight Performance Tools. *Competence in High Performance Computing 2010*, pages 165–175, 2011.

[101] Usha Upadhyayula and Steve Scargall. Introduction to Persistent Memory Configuration and Analysis Tools Usha Upadhyayula (Intel) Steve Scargall (Intel). 2018.

[102] Usenix. ACPI Specified Components. `https://www.usenix.org/legacy/publications/library/proceedings/usenix02/tech/freenix/full_papers/watanabe/watanabe_html/node4.html`.

[103] Alexander Van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. Persistent memory I/O primitives. *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 2019.

[104] Sudharshan S. Vazhkudai, Bronis R. De Supinski, Arthur S. Bland, Al Geist, James Sexton, Jim Kahle, Christopher J. Zimmer, Scott Atchley, Sarp Oral, Don E. Maxwell, Veronica G.Vergara Larrea, Adam Bertsch, Robin Goldstone, Wayne Joubert, Chris Chambreau, David Appelhans, Robert Blackmore, Ben Casses, George Chochia, Gene Davison, Matthew A. Ezell, Tom Gooding, Elsa Gonsiorowski, Leopold Grinberg, Bill Hanson, Bill Hartner, Ian Karlin, Matthew L. Leininger, Dustin Leverman, Chris Marroquin, Adam Moody, Martin Ohmacht, Ramesh Pankajakshan, Fernando Pizzano, James H. Rogers, Bryan Rosenburg, Drew Schmidt, Mallikarjun Shankar, Feiyi Wang, Py Watson, Bob Walkup, Lance D. Weems, and Junqi Yin. The design, deployment, and evaluation of the CORAL pre-exascale systems. In *Proceedings - International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018*, pages 661–672, 2019.

[105] Daniel Waddington, Mark Kunitomi, Clem Dickey, Samyukta Rao, Amir Abboud, and Jantz Tran. Evaluation of intel 3D-Xpoint NVDIMM technology for memory-intensive genomic workloads. *ACM International Conference Proceeding Series*, 11(19):277–287, sep 2019.

[106] Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, and Weikuan Yu. An Ephemeral Burst-Buffer File System for Scientific Applications. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 0:807–818, jul 2016.

[107] Lawrence WEBBER and Michael WALLACE. Advanced Configuration and Power Interface. Technical Report April, 2015.

[108] Wikichip. Skylake (server) - Microarchitectures - Intel - WikiChip. `https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)`.

[109] Jinfeng Yang, L. I. Bingzhe, and David J. Lilja. Exploring performance characteristics of the optane 3D xpoint storage technology. In *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, volume 5, pages 1–28. Association for Computing Machinery, feb 2020.

[110] Judicael A. Zounmevo, Swann Perarnau, Kamil Iskra, Kazutomo Yoshii, Roberto Gioiosa, Brian C. Van Essen, Maya B. Gokhale, and Edgar A.

Leon. A container-based approach to OS specialization for exascale computing. *Proceedings - 2015 IEEE International Conference on Cloud Engineering, IC2E 2015*, pages 359–364, 2015.

Publications

[111] Brice Goglin and Andrès Rubio Proaño. Opportunities for partitioning non-volatile memory DIMMs between co-scheduled Jobs on HPC Nodes. In *Euro-Par 2019: Parallel Processing Workshops*, volume 11997 of *Lecture Notes in Computer Science*, pages 82–94, Göttigen, August 2019. Springer.

[112] Brice Goglin and Andrès Rubio Proaño. Using Performance Attributes for Managing Heterogeneous Memory in HPC Applications. Submitted, 2021.

[113] Edgar A. León, Brice Goglin, and Andrès Rubio Proaño. M&MMs: Navigating Complex Memory Spaces with Hwloc. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, page 149–155, New York, NY, USA, 2019. Association for Computing Machinery.

[114] Andrès Rubio Proaño. Exposer les caractéristiques des architectures à mémoires hétérogènes aux applications parallèles. In *COMPAS 2020 - Conférence francophone d'informatique en Parallélisme, Architecture et Système*, Lyon, France, June 2020.